

AFRL-IF-RS-TR-2003-103
Final Technical Report
May 2003



APPLICATIONS THAT PARTICIPATE IN THEIR OWN DEFENSE (APOD)

BBN Technologies

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J031

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

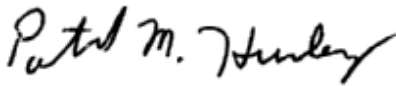
The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-103 has been reviewed and is approved for publication.

APPROVED:



PATRICK M. HURLEY
Project Engineer

FOR THE DIRECTOR:



WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE MAY 2003	3. REPORT TYPE AND DATES COVERED Final Jul 99 – Jul 02	
4. TITLE AND SUBTITLE APPLICATIONS THAT PARTICIPATE IN THEIR OWN DEFENSE (APOD)			5. FUNDING NUMBERS C - F30602-99-C-0188 PE - 62301E PR - H557 TA - 10 WU - 01	
6. AUTHOR(S) Franklin Webber, Partha P. Pal, Michael Atighetchi, Chris Jones, Paul Rubel, Ron Watro, Tom Mitchell, Richard E. Schantz, and Joseph P. Loyall				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) BBN Technologies 10 Moulton Street Cambridge Massachusetts 02138			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/IFGA 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2003-103	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Patrick M. Hurley/IFGA/(315) 330-3624/ Patrick.Hurley@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) The goal of the APOD project was to give software applications an increased resistance against malicious attack even when they run in an environment that is not completely secured. We call any such application "defense enabled". Note that defense enabling is less ambitious than building a secure system: rather than protect the entire system, defense enabling concentrates on the survival and integrity of essential applications, possibly sacrificing other parts of the system to the attacker. Defense enabling also gives priority to some security properties over others: we are much more concerned with defending the integrity of an application's data than its confidentiality. Defense enabling is representative of a relatively recent trend in computer security, often called survivability or 3rd generation security. Several factors distinguish the APOD approach to survivability from others. First, dynamic adaption is a key theme of our approach. Intrusions cause changes in the system, and a survivable system much cope with these changes. As a consequence, defense enabled applications must be very agile and will make use of the flexibility possible in today's dynamic, networked environments. Second, a defense enabled application has a defense strategy that is typically application and mission specific. Such strategies complement and go beyond traditional approaches to security in which protection mechanisms are typically not aware of the applications they aim to protect. Third, defense enabling builds the defense in middleware, intermediate between the application and the networks and operating systems on which the application runs. Defense strategies implemented in middleware can be reused relatively easily in the context of other applications because they are only loosely coupled to the application.				
14. SUBJECT TERMS Survivability, Malicious Attack, Trusted Computing Base, TCB, Protection, Defense, Corrupt, Defense Enabling, Defense Enabled, Self-Stabilizing Software Bus			15. NUMBER OF PAGES 89	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

TABLE OF CONTENTS

1	PROJECT SUMMARY	1
1.1	Identification	1
1.2	Motivation	1
1.3	Project History	3
2	INTRODUCTION	5
2.1	Background	5
2.2	Approach	6
2.2.1	Defending Critical Applications	6
2.2.2	Slowing the Acquisition of Privilege	8
2.2.3	Competing for Control of Resources	11
2.2.4	Classifying Defensive Adaptation	13
2.3	Accomplishments	15
2.3.1	Defense Mechanisms	15
2.3.2	Defense Strategy	17
2.3.3	Validation	18
2.4	Conclusions	22
3	DEFENSE MECHANISMS	22
3.1	Redundancy Management	22
3.1.1	AQuA Replication Management	24
3.1.2	Self-Stabilizing Software Bus	27
3.2	Bandwidth Management	28
3.3	Access Control and Cryptographic Mechanisms	30
3.4	Intrusion Detection	31
3.5	Firewalls	35
3.5.1	Motivation	35

3.5.2	Design and Implementation	36
3.5.3	Related Work.....	37
3.6	Port/Address Hopping.....	38
3.6.1	Motivation	38
3.6.2	Algorithm	39
3.6.3	Design and Implementation	40
3.6.4	Related Work.....	40
3.7	Virtual Private Networks	41
3.7.1	Motivation	41
3.7.2	Design and Implementation	41
3.7.3	Related Work.....	43
3.8	TCP Connection Flood Defense Mechanism	43
3.8.1	Motivation	43
3.8.2	Design and Implementation	44
3.8.3	Related Work.....	44
3.9	ARP Spoof Detection	45
3.9.1	Motivation	45
3.9.2	Design and Implementation	45
3.9.3	Related Work.....	45
3.10	Host Shutdown.....	46
3.10.1	Motivation	46
3.10.2	Design and Implementation	47
3.11	QuO Adaptive Middleware.....	47
3.12	Mechanism Integration	51
4	DEFENSE STRATEGY.....	52
4.1	Local vs. Global Adaptation	52
4.2	A Generic Strategy	53
4.2.1	Outrunning.....	54
4.2.2	Containment	54
4.2.3	Flood prevention and trace back.....	54

4.3	Defense Policy Options.....	55
4.3.1	Outrunning.....	55
4.3.2	Containment	55
4.3.3	Flood prevention and trace back.....	56
5	VALIDATION	56
5.1	In-House Testing.....	56
5.1.1	Experiment Setup	56
5.1.2	Attacks.....	57
5.1.3	Defense Strategies	59
5.1.4	Execution & Data Analysis	60
5.1.5	Lessons Learned from APOD In-House Testing	61
5.2	Red Team Experiments.....	61
5.2.1	Planning.....	62
5.2.2	Execution.....	67
5.2.3	Data Analysis of Primary Metrics	71
5.3	Discussion.....	74
5.3.1	Contrasting In house and Formal Red Team Evaluations.....	74
5.3.2	Ideas for Next Generation Evaluation Experiments	75
6	CONCLUDING REMARKS	78
7	REFERENCES:	80

LIST OF FIGURES

Figure 1 - Object Replication in AQuA.....	25
Figure 2 - Address / Port Hopping via Tunnels and/or NAT Gateways.....	39
Figure 3 - Use of IPsec to provide WAN VPN functionality	41
Figure 4 - TCP Connection Flooding Defense	44
Figure 5 - Host Shutdown triggered by coordinated policy.....	46
Figure 6 - Inter object interaction in Distributed Objects Computing (DOC)	48
Figure 7 - Interposition of QuO Code Generated components in the DOC inter-object interaction.....	49
Figure 8 - Pattern for integrating a mechanism	51
Figure 9 - APOD-1 Attack Plan	64
Figure 10 - APOD-2 Network Topology.....	65
Figure 11 - Time to Denial for Live Attacks	72
Figure 12 - Time to Denial for Scripted Attacks	73
Figure 13 - Attack spoof_block_arp4.....	74

LIST OF TABLES

Table 1 - A Classification of Defensive Adaptations	14
Table 2 - Examples of IDS systems.....	32
Table 3 - Operating Regions of a sample Defense Contract.....	50
Table 4 - Attacks against ATC	60
Table 5 - Red Team Experiment Milestones	63
Table 6 - Attack Results of APOD-1	68
Table 7 - Attack Results of APOD-2.....	70

1 Project Summary

1.1 Identification

This document is the Final Report for the project entitled “Applications that Participate in their Own Defense” (APOD). This project was a 3-year effort by BBN Technologies of Cambridge, Massachusetts, funded by the US Department of Defense Advanced Research Projects Agency (DARPA), under contract F30602-99-C-0188 beginning in July 1999.

This Final Report describes the problem that motivated the effort, the main approaches investigated, the technology developed, lessons learned from both successful and unsuccessful approaches, and some tasks that need to be done to continue this research direction. This section summarizes the motivation and the history of the project. The remainder of the report provides more detail.

1.2 Motivation

Malicious attacks against computer systems are becoming more common and more damaging. A *malicious attack* against computer system X is a sequence of actions taken by a user of some computer system Y to alter or deny data processing in X in ways that are not authorized. If X and Y are the same, the user is called an *insider* (i.e., he is authorized to use X but not to attack it); if they are different, he is called an *outsider* (e.g., he attacks X over the Internet from Y). Malicious attacks are becoming more common because computers are becoming more interconnected. Malicious attacks are becoming more damaging because computers are increasingly relied on to provide essential data and services, so unauthorized changes to these data and services are less tolerable.

Secure computer systems protect against malicious attacks by making such attacks very difficult or impossible to carry out. Unfortunately, building a completely secure computer system is now known to be exceedingly difficult, an ideal that is very costly during both development and maintenance, and that involves significant burdens for users. Practical alternatives are needed.

The goal of the APOD project was to give software applications an increased resistance against malicious attack even when they run in an environment that is not completely secured. We call any such application “defense-enabled”. Note that defense enabling is less ambitious than building a secure system: rather than protect the entire system, defense enabling concentrates on the survival and integrity of essential

applications, possibly sacrificing other parts of the system to the attacker. Defense enabling also gives priority to some security properties over others: we are much more concerned with defending the integrity of an application's data than its confidentiality. Note that this priority is implicit in our definition of "malicious attack", which does not mention the possibility that the attacker might steal confidential data.

The APOD approach is, in brief, to respond and adapt to the effects of a malicious attack while it is in progress. If the defense-enabled application continues correct processing in spite of the attack, we consider the defense to be successful. Even if the attacker eventually is able to damage the application, the defense will have been successful if it allowed time for system administrators to detect the attack and respond to it, perhaps by physically blocking the attacker's continued access to the system. Any technique for extending an application's life while under malicious attack is potentially part of defense enabling.

Defense enabling is representative of a relatively recent trend in computer security, often called *survivability* or 3rd generation security. The 3 generations can be summarized as follows:

- 1st generation security aimed to protect systems completely with software and hardware mechanisms that could not be circumvented. This approach has proved to be quite costly and inflexible.
- 2nd generation security acknowledged that implementations of 1st generation protection are imperfect, allowing attackers to circumvent security mechanisms. The 2nd generation aimed to detect attacks, allowing system operators to respond.
- 3rd generation security uses both 1st and 2nd generation techniques but adds automated defenses that respond to an attack. These defenses can respond to the *effect* of attacks even when the attacks themselves are not detected with 2nd generation techniques.

The 3rd generation has proved to be necessary in modern computer systems, which typically rely on "commercial off-the-shelf" (COTS) components whose security characteristics are not known and that interconnect and interoperate with many other systems. The use of insecure COTS means that the 1st generation approach is impossible. The openness of these systems results in the constant discovery and use of new attacks. When 2nd generation techniques fail to detect these new attacks, 3rd generation techniques are necessary.

Several factors distinguish the APOD approach to survivability from others. First, dynamic adaptation is a key theme of our approach. Intrusions cause changes in the system, and a survivable system must cope with these changes. As a consequence, defense-enabled applications must be very agile and will make use of the flexibility possible in today's dynamic, networked environments. Second, a defense-enabled application has a defense strategy that is typically application and mission-specific. Such strategies complement and go beyond traditional approaches to security in which protection mechanisms are typically not aware of the applications they aim to protect. Third, defense enabling builds the defense in middleware, intermediate between the application and the networks and operating systems on which the application

runs. Recent developments in middleware allow us to develop adaptive defenses quickly. Advanced middleware allows us to base these defenses on information from layers both above and below, i.e., from the application and from the infrastructure, as well as allowing us to control and dynamically adjust aspects of each layer. Defense strategies implemented in middleware can be reused relatively easily in the context of other applications because they are only loosely coupled to the application.

1.3 Project History

As part of the APOD project, we created an APOD Toolkit that allows developers to defense-enable their own applications. We released several versions of this Toolkit, each with greater capability and reflecting greater insight into the problem domain:

- In December 1999, we released version 0 of the Toolkit, containing two very simple distributed applications with adaptive defenses. This release was intended as a proof-of-concept, showing that simple adaptive defenses were possible and could be built easily.
- In August 2000, we released version 1 of the Toolkit. By this time, we had outlined a basic set of defense mechanisms that could be used as the building blocks for defense enabling. This set, which is not intended to be comprehensive, is described in Section 2.2.4. Toolkit version 1 included early versions of many of these mechanisms and some simple examples of how to apply them.
 - One key problem the APOD Toolkit needed to address was integration of different mechanisms in a single application. Because of an integration problem, Toolkit version 1 did not include a key defense mechanism: distributed access control. By November 2000, we had solved this problem and released an update, version 1.1.
- In September 2001, we released version 2 of the Toolkit. This version included both new and updated defense mechanisms and some new applications for demonstration. By this time, we had begun to concentrate on the problem of defense strategy, i.e., how to use various defense mechanisms in concert to achieve the best defense. The version 2 applications were based on some simple defense strategies but these strategies were not well developed.
- We released the final APOD Toolkit, version 3, in August 2002. Version 3 improved on version 2 in an important way: one of the defense-enabled applications in the Toolkit had been the subject of two Red Team experiments and had thus received more scrutiny than any previous defense-enabled application. In response to this scrutiny, we improved existing defense mechanisms, added some new ones, and repeatedly refined the defense strategy used for the experiments. All results from these efforts were included in the final Toolkit.

Toolkit versions 2 and 3 were released, with government approval, under an open-source license.

Validation of the APOD approach was an important part of the project. There were two main validation activities:

1. We conducted in-house testing of many of the defense mechanisms in the Toolkit. This testing was done with Toolkit version 1.1 in the autumn of 2000. The mechanisms were tested individually rather than as part of a combined defense strategy. The goal was to discover problems with the mechanisms relatively early in the project and use these discoveries to improve later versions. In early 2001, we delivered a report of test results to the government.
2. As mentioned previously, APOD was the subject of Red Team experiments and was one of the first projects in DARPA's Fault-Tolerant Networking program to undergo this kind of intense validation. The Red Team was provided by Sandia National Laboratory and the experiments were organized by another group at BBN (the "White Team"). The experiments were developed in the autumn of 2001 and carried out in the first half of 2002. These experiments provided a much broader test of APOD than in-house testing, including tests of an overall defense strategy that combined most of the APOD mechanisms.

Our conclusions in this report, for example in Sections 2.3.3 and 2.4, rely heavily on the results and our observations of Red Team experiments.

We described the APOD approach and our results in several papers:

- "Open Implementation Toolkit for Building Survivable Applications", Partha Pal, et al., Proceedings of DISCEX I, January 2000, explains how we intended to use the QuO adaptive middleware as a framework for integrating defense mechanisms (and we did; see Section 3.11).
- "Building Adaptive and Agile Applications Using Intrusion Detection and Response", Joseph Loyall, et al., Proceedings of the Network and Distributed System Security Symposium (NDSS), February 2000, shows adaptive response used for defense.
- "Defense-Enabled Applications", Franklin Webber, et al., Proceedings of DISCEX II, May 2001, explains the concept of "defense enabling" in general. An updated version of this paper forms the bulk of Section 2 of this report.
- "Defense Enabling Using Advanced Middleware: An Example", Partha Pal, et al., Proceedings of MILCOM, October 2001, shows how to apply the concept of defense enabling.
- "Middleware Policies for Intrusion Tolerance: A Position Statement", Franklin Webber, et al., Workshop on Dependable Middleware-Based Systems (part of Dependable Systems and Networks Conference), June 2002, lists the key issues we have identified in formulating a good defense strategy.
- "Experimental Evaluation of Survivability: The APOD Experience", Partha Pal, et al., submitted to DISCEX III, reports on the lessons learned from Red Team validation of APOD.

We demonstrated the APOD technology in several venues:

- DISCEX I, January 2000, in Hilton Head, SC;
- Quorum technology exposition, February 2000, in Washington, DC;
- FTN PI meeting, January 2001, in St. Petersburg, FL;
- DISCEX II, June 2001, in Anaheim, CA;
- FTN PI meeting, August 2001, in Colorado Springs, CO.
- FTN PI meeting January 2002, in San Diego, CA.

Finally, we documented the details of the APOD work in this report. Section 2 describes the approach in more detail. Section 3 is a full description of APOD defense mechanisms, and Section 4 of APOD defense strategies. Section 5 discusses validation. Section 6 offers some directions for future research.

2 Introduction

2.1 Background

Ideally, one defends a computer system against malicious attack by identifying, in a security policy, what one wants to protect and then by implementing that protection in hardware and software. The implementation is called a trusted computing base (TCB) [42]. The TCB is trusted not to violate the security policy itself¹, and, in most systems, it is also trusted to prevent other, possibly malicious, software from violating the policy. In a distributed system, the TCB usually includes key parts of the operating systems that run on network hosts and of the network communication paths between these hosts.

In practice, many computer systems today have no such trusted computing base. Many others have a design for a TCB but its implementation is seriously flawed. There are several reasons for this situation:

- It is hard to keep the TCB for a complicated system simple enough to warrant trust.
- It is hard to modify the TCB while maintaining trust, because even simple changes to the TCB can have unforeseen effects that undermine its protection.
- It is hard to redesign an existing system to create a TCB if none was planned for originally.

In fact, many of the world's computer systems today run operating systems and networking software that are far from the TCB ideal. These systems may lack any security policy, can be damaged using well-known attacks, and therefore cannot be trusted to protect anything. These systems will continue to be used because of the many applications that depend on them, but are unlikely to be redesigned to be more trustworthy.

¹ Technically speaking, the TCB is trusted to violate the policy so long as the effect of that violation is not equivalent to allowing non-TCB software to violate the policy.

Given this situation, one might ask: “What kind of defense is possible for systems that can be accessed by malicious attackers but lack trustworthy operating systems and networking to protect them?”

In principle, the answer is “None”. A determined attacker can, with sufficient work, defeat whatever flawed protection is offered by the operating systems or networking; thus gaining privileges that can be used either to kill the system completely or to corrupt it some other way. Although one might try to protect data using encryption and digital signatures that are computationally infeasible to break [43], when that data is processed by the system it will almost certainly become vulnerable to an attacker with enough privilege. Note that encrypted data is worthless unless it is decrypted at some time, and it can be read at that time by a privileged attacker. Also note that digitally signed data must be re-signed when it is modified, and an attacker who gains the privilege to re-sign data can forge new, corrupt data as well.

In practice, though, an attacker may not have the skill, perseverance, preparation, or time needed to carry out the attacks that are possible in the worst case. Some attackers rely on prepackaged attack “scripts” and do not have the skill to repair the scripts if they fail. An attacker who meets unexpected obstacles may look elsewhere for easier targets rather than persevere in an attack. An attacker who is not prepared in advance to circumvent the protection in a specific system will be more likely to trigger intrusion detection alarms [44]. In any case, the more time an attacker takes, the more vulnerable he is to being detected and stopped by system administrators.

In summary, system protection is not perfect, but attacks and attackers aren't either.

This report makes a distinction between *protection*, which seeks to prevent the attacker from gaining privileges, and *defense*, which includes protection but also seeks to frustrate an attacker in case protection fails and the attacker gains some privileges anyway. Protection mechanisms are static and proactive; defense mechanisms enhance and complement the protection mechanisms with a dynamic strategy for reacting to a partially successful attack. Both protection and defense aim to keep a system functioning, but protection tends to be all-or-nothing, either it works or it doesn't, whereas defense enables a range of possible responses, some more appropriate and cost-effective than others for a particular kind of attack.

2.2 Approach

2.2.1 Defending Critical Applications

The goal of defense is the correct functioning of one or more *critical* applications. These applications are critical in the sense that the functions they implement are the main purpose of the computer system on which they run. Defending other applications in the same environment is not a primary goal. Neither is defending the application's environment itself, e.g., the operating systems and networks that support the critical applications. Defending the environment is important only so far as it helps to defend the critical applications themselves.

We say that an application that does not function correctly is *corrupt*. A corrupt application might deliver bad service or it might fail to deliver service at all. The goal of defense, then, is to prevent or significantly delay corruption of critical applications.

An application can become corrupt due to various causes:

- either because of an accident, such as a hardware failure, or because of malice;
- either because flaws in its environment cause a loss of protection that allows it to be damaged or because flaws in its own implementation cause it to misbehave.

The main concern of this project has been corruption that results from a malicious attack exploiting flaws in an application's environment. We argue that this is by far the most likely cause of corruption and so the other causes will be neglected in this overview. This assumption is reasonable because:

- Malicious attacks, which are directed and intentional, are far more effective in corrupting an application than accidents, which happen randomly.
- Flaws in the application's implementation can be corrected more easily than flaws in the application's environment, and the latter are likely to be better known to attackers and exploited by them.

Note that we are assuming we can modify or extend the design and implementation of the critical applications. This is in sharp contrast with the design and implementation of the environment, which we assume is almost completely beyond our control. In other words, we must live with flaws in the environment but, because our goal is defending critical applications, we will expend the effort to make those applications much more trustworthy than the operating systems and networks on which they depend.

We say that an application is successfully *defense-enabled* if there are mechanisms to cause most attackers to take significantly longer to corrupt it than would be necessary without the mechanisms. In other words, an attacker must not only defeat protection mechanisms in the environment, he must spend additional time defeating defense mechanisms added to the application.

The central factor in both attack and defense is *privilege*. An attack succeeds when the attacker gains privileges that allow him to corrupt some critical application. Defense enabling, therefore, must succeed

either by keeping the attacker from gaining such privileges or by keeping him from using those privileges effectively. Defense enabling, therefore, can be divided into two complementary goals:

- The attacker's acquisition of privileges must be slowed down. How this can be done is the topic of Section 2.2.2.
- The defense must respond and adapt to the privileged attacker's abuse of resources. Mechanisms for doing this are the topic of Section 2.2.3.

Both goals are important. The first one makes the protection in the application's environment last longer. The second one makes the attacker work harder to use newly gained privileges to corrupt a critical application. Because we have assumed that a determined attacker cannot be delayed indefinitely, both goals are needed for defense.

Defense enabling is organized around the application to be defended rather than around the operating systems and networks that support it. This follows simply because the application can be modified whereas the environment, for the most part, cannot. Section 2.2.3 explains that many defense mechanisms will tend to be placed into middleware [45], which is not part of the environment (in the traditional sense we have defined it here) but is still separate from the application's functionality. This separation keeps the defense mechanisms from complicating each application's design and allows for easy reuse in multiple applications.

2.2.2 Slowing the Acquisition of Privilege

Defense enabling depends on slowing the spread of privilege to attackers. To see this, note that if privileges could be gotten instantly, the attacker could immediately grab all the privileges needed to stop all application processing and thus to deny all service. No defense would be possible against this unlimited attack.

To help slow the spread of privilege, we divide the system into several *security domains*, each with its own set of privileges. The intent is to force the attacker to take more time accumulating the privileges he needs to corrupt applications. This will be true if:

- Each critical application has parts that are intelligently distributed across many domains so that privilege in a set of several domains is needed to corrupt it. This distribution of parts will be discussed in Section 2.2.3.

- The attacker cannot accumulate privileges concurrently in any such set of domains. This constraint will be discussed later in this section.

A security domain may be a network host, a LAN consisting of several hosts, a router, or some other structure. The domains are chosen and configured to make best use of the existing protection in the environment to limit the spread of privilege. The domains must not overlap; for example, if the domains are sets of hosts then each host is in exactly one domain.

Each security domain may offer many different kinds of privilege. The following hierarchy is a minimal set that is typical in many domains:

- **anonymous user privilege:** allows interaction with servers in a security domain only via network protocols such as HTTP that do not require the client to be identified;
- **domain user privilege:** allows access only to a well-defined set of data and processes in one particular security domain (e.g., the user must “log in” to get this access);
- **domain administrator privilege:** allows reading and writing of any data and starting and stopping any processing in one particular security domain (e.g., “root” privilege on Unix hosts).

This hierarchy is listed in order of increasing privilege. Each of these privileges subsumes all the previous ones.

To increase the protection of critical applications we create a new kind of privilege in each domain:

- **application-level privilege:** allows interaction with a defense-enabled application using application-level protocols (e.g. CORBA calls that query the application or issue commands).

An attacker with application-level privilege would find it easy to control, and thus corrupt, an application, so defense enabling must make it hard for an attacker to get this privilege.

Application-level privilege is a key part of defense enabling. It differs from other kinds of privilege in that

- it is not part of the environment but is created specifically to defend an application;
- it uses cryptographic techniques (which will be described later);
- it does not subsume any of the other kinds of privilege and it is not subsumed by any of them.

In particular, gaining domain administrator (“root”) privilege does not guarantee application-level privilege; this will be explained shortly.

A malicious intruder will often attack a critical application by collecting the privileges needed to damage its integrity or to stop it from providing service. Using the set of privileges just listed, there are three ways for an attacker to gain new privileges:

- by converting domain or anonymous user privilege into domain administrator privilege (e.g., exploiting bugs in trusted services, such as `sendmail`, that have domain administrator privilege already);
- by converting domain administrator privilege in one domain into domain administrator privilege in another (e.g., using “root” in one domain to log in as “root” in another);
- by converting domain administrator privilege into application-level privilege (e.g., using “root” privilege to invoke unauthorized application commands).

The attacker must be slowed down or prevented from gaining new privileges in each of these ways. How to do this will depend on the nature of the domains and therefore no generally applicable rules can be given. However, the common case at issue today is security domains that are sets of network hosts. The following discussion applies to that case.

First, the attacker may try to convert domain or anonymous user privilege into domain administrator privilege by exploiting operating system security flaws. As explained in Section 2.1, we assume this will always be possible. We also assume that it takes some time, possibly only a matter of minutes, but it is not instantaneous. The time it takes can be maximized by configuration of hosts and firewalls, for example, by applying the latest operating system patches, disabling or blocking unnecessary network protocols, and making the password file unreadable.

Second, the attacker can be prevented by proper host configuration from converting administrator privilege in one domain into administrator privilege in another. For example, hosts in different domains must not respect each other's privileges. This forces the attacker to start from scratch when trying to gain privilege in each domain. Once having become a domain administrator, the attacker can quickly damage application processes in that domain simply by stopping them. With this privilege, he can bypass the operating system access controls that would normally prevent this damage. This damage, though, is contained because the application is distributed and diversified across many security domains.

Third, a defense-enabled application must use cryptographic techniques to prevent the attacker from gaining application-level privilege. An attacker having this privilege can do more damage than a domain administrator because direct attacks on the application cannot be confined to a single security domain. With application-level privilege, the attacker masquerades as part of the application itself, bypassing its access controls and causing it to behave incorrectly by sending it bogus commands and data, which the application itself propagates across the boundaries between security domains. The following techniques are therefore an essential part of every defense-enabled critical application:

- No application process can be started without authentication, e.g., executables are stored on disk encrypted with passwords known only to authorized users and other application processes;

- All communication between application processes is digitally signed with private keys known only to the application itself and communication uses sequence numbers to prevent replay.

Using these techniques will make it hard for an attacker, even one with domain administrator privilege, to masquerade as part of the application. Assuming the encryption is unbreakable, the attacker will be unable to corrupt the application process' code on disk. Assuming the digital signatures are unbreakable, the attacker will be unable to disrupt communication.

In principle, a domain administrator can gain application-level privilege with enough effort. For example, the administrator can read the core image of a running process, modify it to change the process' behavior, or search it to find the private keys used for digital signatures. This attack could be made harder with techniques for concealing or randomizing the location of data, e.g., passwords, within a core image, but the attack would still be possible. To counter this attack directly, the application must be made to confine application-level privilege, most likely using “Byzantine” fault tolerance techniques [46]. In practice, though, the effort needed for this kind of attack is likely to be much greater than the effort needed simply to kill all application processes in the domain, followed by attacks on other domains.

Finally, the attacker must not be able to gather privileges in many domains concurrently. This constraint means that an attack on an application in many domains cannot go just as fast as an attack on one domain (common-mode failures).

An attack that proceeds sequentially, rather than concurrently, is called a *staged* attack; defense enabling relies on the attacker using only staged attacks. We can either assume that staged attacks are necessary or try to make them so. As a practical matter, most attackers will gather privileges sequentially as they explore a system's infrastructure, so staging may be a reasonable assumption. On the other hand, some attacks can be automated and carried out many times in parallel, in which case staging must be enforced. During the APOD project, we assumed that attacks would be staged and we did not explore techniques for enforcing staging.

This section has shown how defense enabling makes an attacker take longer to collect privileges. The next section shows how this extra time can be used for defense.

2.2.3 Competing for Control of Resources

The traditional approach to computer security treats the attacker and defender asymmetrically: the defender has domain administrator privilege, the attacker does not. The defender is given that privilege initially and

uses that privilege to set up static protection both for critical applications and to maintain the asymmetry, i.e., the attacker must never get domain administrator privilege for himself.

In contrast, defense enabling assumes the attacker may eventually gain domain administrator privilege in some security domains, and in those domains the attacker and defender will be in symmetrical positions. What then? Section 2.2.2 showed how the defender can set up a new kind of privilege at the application level and try to protect it using cryptography. But the defender can also use domain administrator privilege to contest the attacker's control of domains. This section discusses mechanisms to use in that competition for resources.

Defense enabling includes the following tasks:

- **Adding Redundancy:**

Creating multiple security domains is not by itself sufficient to force the attacker to spend more time collecting privileges: if some domain were a single point of failure for the application, the attacker would only need to gain domain administrator privilege in that domain and kill application processes there. Clearly the application must be distributed redundantly across the domains.

The simplest solution is to replicate every essential part of the application and place the replicas in different domains. Doing this turns the problem of defense into a problem of fault tolerance, where a “fault” is the corruption of a single replica by the attacker. The replicas must be coordinated to ensure that, as a group, they will not be corrupted when the attacker succeeds in corrupting some of them. Many protocols exist for fault tolerant replica coordination [47].

Note that by creating and enforcing application-level privilege we may be able to simplify the fault tolerance problem to be solved. If the attacker cannot gain application-level privilege then application replicas will, at worst, crash when corrupted, and so it will not be necessary for the application to use more expensive protocols that protect against “Byzantine” corruption [46]. On the other hand, if the attacker can gain application-level privilege, such protocols are needed.

Also note that replication is only one of various ways to add redundancy to an application.

- **Monitoring:**

Incorporating intrusion detection systems (IDSs) [44] will be a part of this task, to collect data at the infrastructure level about possible attacks. Data collected at the application's level is also desirable, though, because it can give a more comprehensive view of the nature of the attack and more insight

into potential remedies, and because it is more relevant to the needs of the application. Two kinds of monitoring are important at the application level:

- **Quality-of-Service (QoS):** whether the application is getting the QoS it needs from its environment and whether it is providing the QoS required by its users. A decrease of either QoS measure is an indication of a possible attack.
- **Self-checking:** whether the application continues to satisfy invariants specified by its developers. A violation of such invariants is an indication that the application may be corrupt, possibly because the attacker has gained application-level privilege.
- **Counterattacking:**

If the source of an attack can be diagnosed with high confidence, resources can be denied to the attacker, for example, by killing the attacker's processes, denying the attacker bandwidth, or blocking communication from hosts running corrupt processes.
- **Adapting:**

If the attacker denies resources to a critical application, for example by killing application processes or flooding communication channels, the application must try to adapt to restore the QoS it needs. There is a wide variety of possible adaptations. The next section describes a classification scheme for defensive adaptations and gives several examples.

2.2.4 Classifying Defensive Adaptation

An application's defense will use one or more kinds of adaptation to counter a particular attack. This section classifies, in several dimensions, a basic set of potential adaptations.

In one dimension, shown vertically in Table 1, adaptations differ by the level of system architecture at which they work. At the highest level, an application can choose to change its own behavior in the face of an attack, either finding an alternate way to proceed or degrading its service expectations. At the next lower level, the application can use QoS management support to try to make its environment offer the QoS it needs. At the lowest level, the application uses services from the operating system and network level to counter the attack, for example by changing details of how application components communicate.

In another dimension, shown horizontally in Table 1, adaptations differ by how aggressively the attack can be countered. At best, the attack can be defeated, i.e., the effect of the attack on the application can be

completely canceled. Second best is for the application to work around the attack, avoiding its effects. Finally, if the attack can neither be defeated or avoided the application can make changes to protect against similar or recurring attacks in the future.

Although Table 1 shows at least one kind of adaptation for each of the nine possible boxes, the set of adaptations is not intended to be comprehensive: undoubtedly others can be invented or would be available with specific operating systems. There may also be other useful categories; for example, the table does not show any “honeypot” defenses in which an attacker is lured into wasting effort on a decoy. In spite of these caveats, though, this set of adaptation mechanisms offers a useful variety of options for creating a strategy for responding to attacks.

	Defeat Attack	Work Around Attack	Guard Against Future Attack
Application Level	Retry failed request	Redirect request; degrade service	Increase self-checking
QoS Management Level	Reserve CPU, bandwidth	Migrate replicas	Strengthen crypto, access controls
Infrastructure Level	Block IP sources	Change ports, protocols	Configure IDSs

Table 1 - A Classification of Defensive Adaptations

A third dimension for classifying adaptations is according to the kinds of attack they work against. In Table 1, essentially two broad kinds of attack are countered:

1. Direct attacks against the application, for example by disrupting the communication between its parts;
2. Indirect attacks, in which resources the applications need are denied.

Direct attacks are countered by the mechanisms working at the application level, plus the use of encryption. An indirect attack might be countered by any of the mechanisms in the table but generally lower-level mechanisms can be more focused. For example, configuring a firewall to block packets from a particular source is a highly focused defense, but one that needs detailed information about the attack to have been collected first. At the QoS level, flooding the network can be countered by bandwidth reservation, over-consumption of CPU by scheduling and priorities, crashing of a node running an application component by migrating the component elsewhere, and relatively privileged operations can be disabled using access control if there is a high risk that they might be used maliciously.

A fourth dimension for classifying defenses is whether a mechanism can be used for protection from attack as well as for response to attack, or just for response alone. Mechanisms in the table's right-hand column, plus CPU and bandwidth reservation, can be used for protection. Why not always turn these strategies “on”

for best protection? Because some of these defenses, e.g., an IDS configured to be very sensitive to attacks, have significant costs and so need to be used sparingly, and others, such as disabling highly privileged operations, impede the normal functioning of the system and so should be used only when necessary.

Incorporating many or all of these adaptation mechanisms into a single application can greatly complicate the application's design. Fortunately, every one of these mechanisms is orthogonal to an application's functionality, i.e., the application should compute the same results regardless of whether or how many defense adaptations have been used. In other words, every one of these adaptations changes *how* an application computes its results, not *what* results are computed. This orthogonality allows the design of defenses to be separated from the design of functionality.

It is natural to separate the design of functionality from the design of defenses by putting the latter into middleware [45]. The functionality can be designed first, then a strategy for defensive adaptation added later. Ideally, the defensive strategy and the mechanisms it uses would be reusable in many different applications, but this is not always possible. For example, access controls are specific to an application, and self-checking of application invariants will depend on application-specific data structures. These mechanisms seem to be exceptions, though: most of the other mechanisms in Table 1 are reusable across applications.

2.3 Accomplishments

The APOD project had three main accomplishments, overviewed in this section and described in more detail in the rest of the report:

1. Motivated by the analysis of Section 2.2, we implemented a variety of defense mechanisms.
2. To achieve the best defense possible, we explored strategies for coordinating the various mechanisms.
3. Using Red Team experiments, we began to measure the overall value of defense enabling.

2.3.1 Defense Mechanisms

Section 2.2 described the need for a variety of defense mechanisms. The APOD project implemented most of the mechanisms described, and combined these mechanisms in an open-source APOD Toolkit [48].

The next few paragraphs list the kinds of mechanisms we needed and points the reader at later sections in the report where more detail about those mechanisms can be found.

First, Section 2.2.2 defines application-level privilege and outlines how to make it difficult for the attacker to get it. We used application-level, cryptography-based access control, described in Section 3.3, to protect a critical application’s communications. To prevent traffic analysis, we used VPNs, described in Section 3.7.

We did not completely implement application-level privilege. In particular, we provided no means for automatic, secure start-up of new application processes, which will be necessary to support replication management without user intervention. This lack will be remedied in another BBN project, “Intrusion Tolerance by Unpredictable Adaptation (ITUA)” [49], which aims to provide a more complete set of tools for defense enabling.

Second, Section 2.2.3 describes the need for replication management. Our implementation, using crash-fault-tolerant protocols, is described in Section 3.1. A more complete implementation, needed to block attacks in which the attacker gains application-level privilege, would use Byzantine-fault-tolerant protocols. The ITUA project also aims to provide this.

Section 2.2.3 also notes the need for monitoring mechanisms. Our use of intrusion detection systems (IDSs) is explained in Section 3.4, but we also depend on QoS monitoring built into other mechanisms such as replication and bandwidth management. Monitoring and detection of TCP connection floods is a key part of the mechanism described in Section 3.8, and detection of ARP cache poisoning is explained in Section 3.9.

Countermeasures to attack and isolation of the attacker are also important. We incorporated bandwidth management (Section 3.2), dynamically reconfigurable firewalls (Section 3.5), and automatic host shutdown (Section 3.10).

Finally, Section 2.2.4 classifies mechanisms according to the adaptation they provide. Table 1 in that section classifies adaptation mechanisms at the application, middleware (QoS), and infrastructure layers. The APOD Toolkit contains no mechanisms for application-level adaptation because such mechanisms tend to be application-specific and thus not easily reusable. We implemented all the other mechanisms listed in Table 1 except for adaptive access control policies (the access control mechanism described in Section 3.3 does allow dynamic policy change but we did not incorporate this feature into the APOD Toolkit). In particular, migrating replicas is part of the replication management mechanism of Section 3.1 and blocking IP sources is part of the dynamic firewall mechanism of Section 3.5. A mechanism for dynamically changing communication ports is described in Section 3.6.

For the reasons explained in Section 2.2.4, we integrated the various defense mechanisms using middleware. The framework we used for that integration, the QuO adaptive middleware, is described in Section 3.11.

2.3.2 Defense Strategy

A defense strategy answers the questions “To counter this attack, which possible adaptations should be used? When should they be used?” These questions can be hard to answer even when only one defense mechanism is being used. In APOD, we usually needed to answer them for a collection of interacting defense mechanisms.

These questions have definite answers only when the goal of the defense strategy is clear. For APOD, we sometimes took the goal to be maximizing the time to failure of the application under the worst anticipated attack, but sometimes we only asked that the defense strategy increase the time to failure under typical attacks. Neither of these goals is precise (e.g., they do not answer “which attacks are anticipated?” or “which are typical?”), but they were adequate for our purpose, which was to begin to explore the space of defense strategies. More precise definitions of “survivability” and more formal analyses of defense strategies are being carried out under the ITUA project [49].

Early in the APOD project, we thought that the choice of defense strategy would be highly dependent on the application requirements. So, for example, an application that consumes a lot of bandwidth would of course use bandwidth management as part of its defense. This idea is especially compelling for defense mechanisms at the application level in Table 1. For example, a strategy for gracefully degrading service when under attack is clearly going to depend on the kinds of service the application provides.

Later in the project, though, we found that the defense strategies we chose tended to have a common, reusable form. Probably this tendency was the result of our focus on defense mechanisms at the QoS management and infrastructure levels in Table 1. We concentrated on defense mechanisms that could easily be reused and found it natural to combine them in reusable ways. In retrospect, it is not clear whether we would also have found reusable strategies for application-level mechanisms as well.

Our best defense strategies tend to have two parts:

1. “**Outrun**”: move application component replicas off bad hosts and on to good ones faster than they can be corrupted by the attacker;
2. “**Contain**”: quarantine bad hosts, LANs, and security domains by limiting or blocking network traffic from them and, within limits, shutting them down so they do no further damage.

Each part of the strategy raises several policy issues. For “outrunning”:

- Where should a new replica be placed? Always in a new security domain? Always on a new host? Using an unpredictable algorithm?
- Should the number of replicas per component change under attack? Increasing the amount of replication will slow the application down but will protect against the possibility the attacker has silently infiltrated more domains than the defense is aware of.

“Containment” raises these policy questions:

- Should one host (or LAN or domain) ever quarantine another? Or should containment rely entirely on self-shutdown based on local sensors?
- Is agreement necessary before quarantine? A purely local decision is easier to spoof but a global decision may be blocked by flooding.
- How will the attacker be prevented from spoofing the defense into quarantining all security domains? By limiting the number of quarantined domains or by limiting the rate of quarantining? Should a quarantined domain be reintegrated under some conditions?
- When is a domain, LAN, or host judged to be bad? How much weight should be given to the source of warnings, whether warnings are repeated, and whether warnings occur in combination?

During the APOD project we gave *ad hoc* answers to these policy questions, some of which are described further in Section 4. We are nearly certain now that some of our policy decisions were less than optimal. We envision future defense enabling toolkits providing a configurable defense based on the answers to these, and other, policy questions.

2.3.3 Validation

The defense enabling approach can be validated in several ways, each of which will be described shortly:

- Modeling;
- Testing;
- Intrusion injection;
- Red Team experiments.

During the APOD project, we used testing and Red Team experiments to try to learn whether defense enabling increases resistance to attack, by how much, and which parts of the defense are weakest. The ITUA project [49] will additionally be using modeling and intrusion injection for validation. We begin this section with a description of each of these techniques. Ultimately, we would like to know which of the techniques gives the best insight into defense enabling and the most accurate measurements of its value.

Validation by modeling involves constructing a mathematical object that encapsulates key features of the system, then reasoning about the properties of this object. The model must include some features of the application being defended, the defense mechanisms and defense strategy, and assumptions about what the attacker can and cannot do. In general, we do not *know* for certain what the attacker cannot do, so the model's assumptions about the attacker are necessarily statements about attacks that are unlikely or hard to carry out. Whenever possible, we base such assumptions on observation of real-world attacks.

Reasoning about models will allow conclusions such as:

- Conditions under which one management strategy is better than another;
- Estimates of how much extra survival is gained using a particular strategy;
- Requirements on the environment, such as the quality of intrusion detectors, needed to gain a specified level of survivability.

The other validation methods differ from modeling in that they work with the actual defense-enabled system rather than an abstraction of it.

Validation by testing involves ensuring that the individual defense mechanisms work as expected. This kind of validation is, of course, part of the normal software development process. It is specialized for defense enabling by testing the effects of expected kinds of attack. For example, when using replication management, one expects that the attacker will be able to kill some replicas. One tests the mechanism under this attack to ensure that the mechanism responds correctly, but also to measure the response time and overhead; these measurements will be used in the validation by modeling already described.

Validation by intrusion injection goes beyond ordinary testing by creating situations that might arise during an attack but which the designers believe are impossible for very difficult to arrange. For example, a replica coordination protocol may be designed to reject any message that comes from a replica known to have behaved incorrectly in the past and therefore likely to have been damaged by the attacker. Intrusion injection would be used to “inject” such messages at a point in the protocol where they should not exist and thus test what would happen in this seemingly impossible situation.

Finally, **validation by Red Team experiment** goes beyond ordinary testing by seeking attacks not foreseen by the designers. Experience shows that a system's designers (also called the Blue Team) are not the best people to find new kinds of attack against the system they designed. Needed is a different group of people, (the Red Team), who will find it easier to think about the system in new ways and who are skilled in finding system vulnerabilities.

A Red Team experiment is a test in which attacks on the system are tried and the results observed. The Red Team is given:

- The system, including defense-enabled applications;
- A set of goals, called “flags” (e.g., “cause the system to deny service for 15 minutes”)
- A set of constraints, also called “rules of engagement”, which prohibit the Red Team from attacking known vulnerabilities that could be closed in known ways (e.g., “killing client C is prohibited because in a real system it will be protected by some other mechanism not reproduced in the experiment testbed”).

The Red Team then tries to “capture the flag” without going “out of bounds”, i.e., without violating the rules of engagement.

We ran two Red Team experiments against a defense-enabled application. The Red Team was provided by Sandia National Laboratory and funded by DARPA. The application was a simple video image server, written specifically for this experiment, in which clients use a broker to locate an appropriate image server, then get images directly from that server. The defenses included replicating the broker, and other mechanisms described in Section 3. The primary flag was denying access to the broker replicas. The defense-enabled application was run in a testbed providing 14 hosts on 4 LANs for application processes, plus a number of other hosts needed for routing and control of the network interconnections.

The Red Team used combinations of the following basic attacks against us, starting with “root” (i.e., system administrator privilege on Unix) on a single host:

- Spoofed scans to cause the defense to (mistakenly) quarantine good hosts;
- ARP cache poisoning to isolate hosts or partition the network;
- TCP connection floods to consume ports;
- TCP connection resets and bad traffic to disrupt communication;
- Network flooding to delay or deny service;
- Replay of RSVP traffic or injection of bogus traffic to stress the bandwidth reservation mechanism.

We observed the following outcomes from both Red Team experiments:

- The Red Team was always able to capture the flag (i.e., deny service) eventually.
- The Red Team spent significant amounts of time (measured in weeks) learning about the defense mechanisms and significant amounts of time (measured in days) discovering the attacks that captured the flag.
- After the attacks were automated in scripts, the shortest times to capture the flag were 5-10 minutes.
- Attacks that captured the flag always set off numerous alarms soon after the start of the attack and before service was denied.

Also significant were actions that the Red Team did *not* take:

- Stealthy attacks were not attempted. So the Red Team experiments did not tell us anything about the difficulty of capturing the flag without setting off warning alarms.
- No attempt was made to gain privilege on hosts other than the one on which the Red Team was “root”. It was decided in advance that such attacks would mainly tell us whether system administrators had applied the most recent operating system patches and not whether defense enabling increased resistance to attack. Unfortunately, this means the Red Team experiments tell us nothing about whether our replication management defense would be able, in practice, to “outrun” an attacker, i.e., start replacement replicas faster than they can be killed.
- Although it was known in advance that an attack that corrupted the in-memory image of any application process could be used to capture the flag, the Red Team decided not to attempt this attack. Such an attack would work because we used protocols that are only crash fault-tolerant and would therefore not tolerate faults that are arbitrarily malicious, or Byzantine [46]. Because the application and key parts of the defense were written in Java, and because Java moves data structures around in memory, the Red Team judged that other kinds of attack would yield greater effect for less effort. Unfortunately, this means the Red Team experiments tell us nothing about the cost of preparing an attack that would need Byzantine fault-tolerant protocols to defend against.

We draw the following conclusions from these observations:

- Defense enabling will force even highly skilled attackers to work hard to deny service, but the best attacks can be automated and therefore can be applied by people with few skills.
- A defense-enabled application increases survival time (because an attacker with “root” can kill a non-defense-enabled application immediately, so 5-10 minutes is a significant increase), but we expect that survival times of 20-30 minutes will be needed as a practical matter to give human operators time to intervene in an attack.

So our implementation of defense enabling has survival value but is not yet good enough for practical purposes.

The APOD Red Team experiments did not settle the key question about the value of defense enabling: In a race between ever-cleverer attacks and ever-improving defense, which side tends to win? If the attack were to win, survival times would likely shrink to near zero. If the defense were to win, survival times would likely grow to 20-30 minutes or longer. Our series of two Red Team experiments was not long enough to provide an answer. Between the experiments, we added new mechanisms (e.g., a defense against the TCP connection flood used in the first experiment) and we inadvertently introduced new flaws, some of which were exploited by the Red Team in the second experiment. Only a longer series of experiments could determine whether the evolving defense would converge to a stable set of mechanisms, comprehensive enough to block all quick Red Team attacks and trustworthy enough not to be a vulnerability itself.

We argue on general grounds that the mechanisms of defense enabling are simple enough that a trustworthy implementation of them is practical. But we did not demonstrate such an implementation in the APOD project. Certainly the defenses we have demonstrated are conceptually simpler than most general-purpose operating systems; therefore, creating trustworthy implementations of them should be easier than the problem of creating trustworthy operating system protection.

2.4 Conclusions

The APOD project showed that defense enabling can increase an application's resistance to malicious attack in an environment that offers only flawed protection. This increased resistance means that an attacker must work harder and take more time to corrupt the application. This, in turn, means greater survivability for the application on its own and an increased chance for system administrators to detect and thwart the attack before it succeeds.

The APOD project, however, did not produce a practical implementation of the defense enabling concept. The set of mechanisms is not quite complete, better understanding of strategies is needed, and Red Team experiments show that survival times are still too short by a factor of 3 or 4.

We think that future work on defense enabling should begin with the following topics:

- An analysis of the defense strategy described in Section 2.3.2. Do the best strategies always have these two components (outrun and contain) or are there others? How should the policy issues be resolved?
- Further Red Team experiments with a more stable set of defense mechanisms.

3 Defense Mechanisms

3.1 Redundancy Management

Having alternative means to continue to operate when ongoing attack makes parts of the system unusable is crucial for survivability. Redundancy plays a crucial role in providing the alternatives. Various types of redundancy could be used in defense: redundant hosts, processes, networks, etc., are examples of spatial redundancy, and the ability to redo a computation at a later time is an example of temporal redundancy.

However, simply having redundancy alone is not quite sufficient for survival because the attacker can continue to corrupt or consume the redundant resources. Redundancy needs to be managed in an adaptive

way as part of the overall defense strategy to make it increasingly difficult for the attacker to keep consuming or corrupting redundant resources and/or replenishing the lost resources. In APOD we investigated how redundancy at the process level can be used in defense enabling.

Process level redundancy, in terms of a distributed-object application, means replication of objects. As part of the application's defense strategy, key application objects are replicated and managed. As a result, mechanisms that provide the capability to replicate the objects may be integrated in the defense-enabled application. Such a replication management mechanism may bundle other mechanisms (such as a group communication system) and depending on that different replication management mechanisms offer different properties about the replicas they manage and impose different restrictions on the behavior of the object being replicated. For instance, replicas of a CORBA object in AQuA [25] maintain the virtual synchrony property, using the underlying Ensemble group communication system, and assuming that the object being replicated is deterministic, i.e., always produces the same output in response to the same input and object state. On the other hand, the Self-Stabilizing Software Bus, which is not based on any group abstraction, does not guarantee the synchrony of the replicas it maintains and does not impose the deterministic restriction.

When we first began using object replication in a defense strategy we explored various possibilities of replicating key application objects. We explored how one can start and manage multiple copies of a key application object in an ad-hoc manner within the application: this meant modifying the application but did not require integration of any external replication management mechanism. We also explored the use of fault-tolerance mechanisms like AQuA [25], Rampart [26], SecureRing [27], and Eternal [28], to replicate key application objects. Of these, we implemented and experimented in greater depth with AQuA, and earlier APOD releases demonstrated how AQuA capabilities could be used in defense enabling. During the later part of the project, we focused mostly on the more lightweight replication management known as the Self-Stabilizing Software Bus [29].

There are some general risks associated with using replication in defense. First, the software supporting replication is complex, and therefore has potential undiscovered bugs that attackers can exploit. This is true to some extent, but this speaks more to the implementation of replication mechanism than the concept of using a replication mechanism as a defense mechanism. Second, they introduce overhead and often impose specific restrictions. This criticism has some merit, but this should be viewed as the price for the benefits of replication. It is also often claimed that replication improves the availability and reliability aspect of the system, but degrades the confidentiality aspect—the argument being that by replicating we are increasing the ease of exposing the secret. Other than the fact that the attacker can observe and attack replicas in parallel this argument seems flawed: if confidentiality is important then the non-replicated version of the system must incorporate some confidentiality mechanism, and in theory it should be possible to apply

similar protection to the replicas. In fact, we have shown how encryption and access control can be used in conjunction with replication. Finally, we argue that if the attacker can destroy or corrupt one copy of the critical object, he can easily do the same to all replicas. This argument makes a lot of sense if one claims that replication is a protection or preventive measure. We use it as a defense mechanism with the objective of prolonging the useful life of the system. In that sense, it is always better to have some redundancy. Furthermore, defense enabling is not simply adding replicas, it also includes adaptive management of replicas: moving them around, starting and stopping replicas, incorporating security measures to acquire and transfer privileges among hosts, heterogeneity and deception.

In the remaining part of this section we will describe the use of AQuA and the Self-Stabilizing Bus as defense mechanisms.

3.1.1 AQuA Replication Management

AQuA was developed jointly by researchers at BBN and the University of Illinois under the DARPA Quorum and Information Survivability programs. It provides a process-group abstraction with the property of “virtual synchrony” for communication with group processes. The Ensemble group communication mechanisms underlie the group abstraction and the Maestro interface simplifies the use of process groups with CORBA.

AQuA can be used as a standalone mechanism for tolerating crash, value and timing failures of CORBA objects. AQuA provides an interface for defining: the class of failures to be tolerated; the object to be replicated; the type and number of failures; and a period of recovery after which the system will declare that service cannot be guaranteed if it does not have the required number of replicas (see Figure 1). The AQuA interface also allows one to change the fault tolerance QoS in a limited way. For example, the number of tolerated failures can be changed (say from 2 failures to 3), but the fault model (e.g., CRASH, VALUE) cannot. One also has limited control over where new replicas will be started during recovery from a failure. In addition to these control options, the interface also supports various reporting options through which AQuA can provide details about the failures it observes.

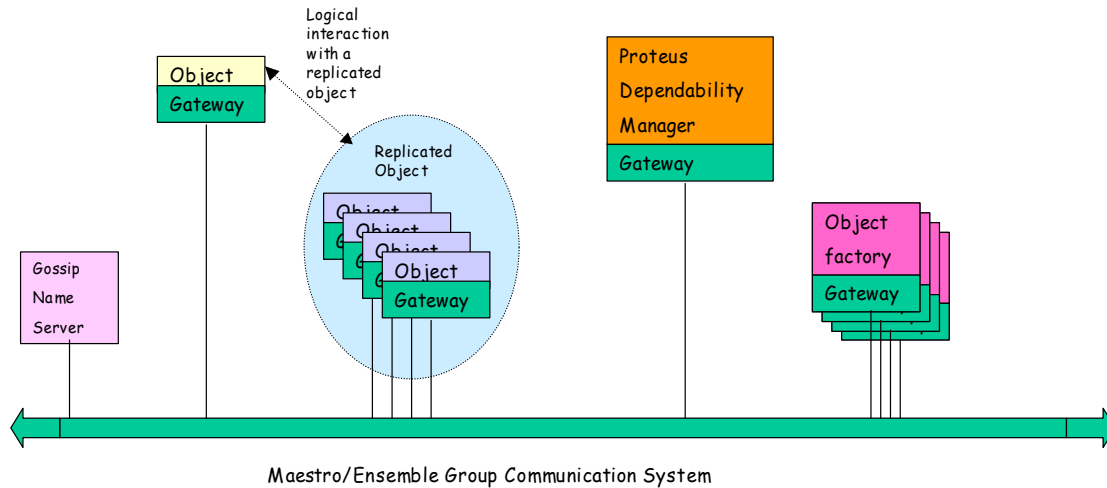


Figure 1 - Object Replication in AQuA

Following are several example strategies for using AQuA's capabilities in defense enabling. Each strategy assumes that the goal is to ensure continued availability of some critical object that is replicated. Note that using AQuA for process level redundancy requires redundant hosts: enough hosts need to be available to run the desired number of replicas. Hosts running replicas are called replication hosts.

Strategy 1: Maintain a constant number of replicas, but monitor the failures. If failures on a particular replication host reach a threshold, abandon that replication host and try to start new replicas on other hosts.

Strategy 2: Maintain a constant number of replicas initially, but if repeated failures are observed increase the number of replicas.

Strategy 3: Maintain a constant number of replicas, but also maintain a standalone replica as backup running on a host that is better secured than others. If repeated failures are observed or AQuA reports that it cannot sustain the requested service, switch over to using the backup. A host better secured than others will be costlier and less freely usable than other hosts, so it would be impractical to secure all hosts at that level.

Using AQuA for defense introduced several risks. We investigated how these risks can be addressed by a) modifying AQuA itself and b) by addressing the risks in the defense strategy. The latter complements the research on combining AQuA with other defense mechanisms in more complicated defense strategies.

The original design of AQuA relied on a gossip server and a centralized manager (known as the Proteus dependability manager). Either of these could become a single point of failure. Responding to our request, the University of Illinois made later versions of AQuA support passive replication of the dependability

manager. This means that if the attacker were to destroy the currently running dependability manager, AQuA would still be able to create a new replica. The gossip single-point-of-failure can be handled by running multiple gossip servers.

AQuA's group abstraction does not have any notion of security: any object wishing to join the group is allowed, any object wishing to invoke any operation on a replicated object is also allowed. We have included some degree of access control in an experimental enhancement of AQuA where OO-DTE (Object Oriented Domain Type Enforcement, an access control mechanism for distributed objects, developed at NAI) and AQuA both needed some modification to co-exist (see Section 3.3). We did not maintain or formally release this version. This issue is likely to be addressed in the context of CORBA security.

AQuA is very sensitive to network level features: if the capacity and throughput is insufficient or there is a change in network security policy, e.g., firewall rules, AQuA replication becomes unstable. This has to do with the design of AQuA and its underlying algorithms that rely heavily on inter-group communications.

We attempted to mitigate these risks at the strategy level by using other defense mechanisms along with AQuA; we had mixed results. By using IDSs along with AQuA we were able to support useful defensive adaptation such as:

Strategy 4: Move replicas off a host, which is flagged as suspect by an IDS. When administrators determine that the host is "clean", inform AQuA that this host is ready to host replicas again.

Strategy 5: Use a combination of IDS alerts and failures reported by AQuA to decide:

- Whether to avoid a particular host or hosts on a particular network segment for replication
- When to switch between replicated and non replicated providers of the same service

Our attempts to use bandwidth management mechanisms such as RSVP with AQuA replication were less successful: adding bandwidth management did not address AQuA's sensitivity to bandwidth and latency fluctuations.

In conclusion, the use of AQuA-based replication as a defense mechanism has its pluses and minuses. Using a replicated Proteus, multiple gossip instances and multiple IDSs one can construct useful defense strategy if other threats like loss of confidentiality, unauthorized access, flooding etc. are taken care of separately.

3.1.2 Self-Stabilizing Software Bus

The Self -Stabilizing Software Bus (often referred to as the “bus” for short) is a mechanism for data publication and process management. Given the information required to start a process (like the location of the executable, the start up parameters) and the desired number of copies of that process, the bus attempts to start and maintain that many copies of the process. A process that is not started by the bus can also “attach” itself to the bus and ask for a handle to an object maintained by the bus. A process attached to the bus can “put” data to the bus and that data will eventually be available to every bus process. The bus is self-stabilizing in the sense defined by Dijkstra: an arbitrary change to the bus data will eventually lead to a stable state in which all functioning bus processes agree on the bus data. Thus the bus is inherently fault-tolerant: some kinds of failure, including crashes of bus processes and corruption of its data, are corrected by the bus mechanism as it stabilizes. One side effect of the bus mechanism is that, in a stable state, every functioning bus process knows the state of the bus (data published to the bus). This fact makes it easy for the bus to manage process replicas: whenever there are too few replicas of one kind of process, a new replica is created. To support replication management, the bus publishes specifications that tell how many processes of each kind are to be maintained and constraints on where new processes can be created. Like any other bus data, these specifications can be changed dynamically and changes will propagate everywhere on the bus.

The basic strategies described in Section 3.1.1 (Strategies 1 through 3) could be implemented using the bus as the replication management mechanism. However, it is important to understand that the bus does not guarantee virtual synchrony for the replicas: it is the responsibility of the object being replicated to make use of the bus features to maintain the state of the replicated object consistently. Because the bus allows one to publish information about processes, it is possible to attach and manage a lot more information with the replicas. This leads to useful adaptation strategies such as:

Strategy 6: Keep track of the network segment on which replicas run, if a replica crashes try to avoid starting the replacement replica in the segment that hosted the crashed replica.

Strategy 7: Use a multi-level or diverse replication strategy where there are different kinds of replicas with different implementations and security-cost characteristics all providing the same functionality. In the absence of any security alert, choose the low-security replicas, and choose the high security replicas otherwise.

One drawback of the bus is its lack of support for maintaining state consistency of replicas. For objects that do not have state, the bus is readily applicable. For stateful objects, the consistency protocol needs to be built at the application level. This is often the source of subtle timing bugs. We have thought about adding

some synchrony support (like locks or circulating tokens) in the bus which will allow implementation of these application level state consistency protocols much easier.

As part of its self-stabilizing mechanism, the bus attempts to heal itself by continuously adjusting the interconnection of bus processes. This provides better performance in a network where bandwidth and throughput cannot always be guaranteed. Furthermore, this allows the bus to deal with network partitioning by forming independent buses in each partition and joining them back when the network partition heals. However, the bus lacks security support to verify that the delayed/excluded processes are not corrupt and does not provide enough support to merge the bus data after a partition heals. This issue needs to be addressed at the application level by the objects that make use of the bus.

We have used the bus in conjunction with other defense mechanism such as OO-DTE [30] access control, RSVP [31] bandwidth management, intrusion detection systems and packet filtering mechanisms. Using OO-DTE with the bus, it is possible to deny unauthorized access to objects maintained by the bus and unauthorized changes in bus data. Using RSVP with the bus lead us to the following defensive strategy:

Strategy 8: Maintain the critical object using the bus, i.e., if it fails start a replacement. In addition, if interaction of a client with the critical object is slowed down because of network flooding, establish a reservation to expedite the interaction. If O2 is replacing O1, and there was a reservation established with O1, tear down the old one and reestablish the reservation between the client and O2.

Using the bus along with intrusion detection systems it is possible to support strategies 4 and 5 described above. In addition, using the bus to publish IDS data, it is possible to coordinate a packet filtering mechanism to respond. Such a strategy is described later in the document in Section 3.

The bus provides a very flexible and lightweight mechanism to do process replication. It is very easy to use, portable and has a small footprint. It works perfectly fine for replicating stateless objects, however, for statefull objects a little more work is required in the form of application level coordination protocols for maintaining state consistency.

3.2 Bandwidth Management

Many network applications need a minimum amount of bandwidth to work effectively. APOD investigated different types of bandwidth management with the goal of allowing APOD strategies to counter network resource exhaustion attacks that can severely hinder applications. Integrating bandwidth management into

APOD strategies allows APOD enabled applications with the ability to request the needed bandwidth during an attack.

There are two bandwidth management models:

- Integrated services (IntServ)[21] and
- Differentiated services (DiffServ)[20].

Both of these models were defined by the Internet Engineering Task Force (IETF) and are described in RFCs. IntServ is a signaled-QoS model where end-hosts signal their QoS needs to the network infrastructure. DiffServ provides QoS by elements that are set up to service multiple classes of traffic with different QoS requirements. We chose a CORBA interface developed by the DIRM [24] project that allows a client to request bandwidth. DIRM is a CORBA wrapper for an underlying resource reservation system developed at Columbia University that implements a specification called Resource ReSerVation Protocol (RSVP).

RSVP can reserve bandwidth along a path for specified traffic. Because the traffic is specified by the source and destination IP addresses and port numbers, RSVP follows the IntServ approach. RSVP is a stateful protocol, meaning that reservation information must be kept on intermediate routers. This allows APOD to reserve bandwidth along a path to and from clients and servers. APOD developed an example that used RSVP and showed that reservations could be made dynamically during attacks that flooded the network. The example contained an image server and a client requesting images from the server. If a flood was detected, a reservation between the server and client is established. If the server dies and another is started, the old reservation was removed and a new reservation from the new server to the client is established.

In addition to integrating with DIRM's RSVP, we established interfaces to use SE-RSVP in APOD (Security Enhanced RSVP)[23] as a RSVP mechanism which is hardened against certain attacks. SE-RSVP consists of a modified version of the University of Darmstadt's RSVP [22] implementation with authentication and access control added. SE-RSVP was tested extensively in the red team experiment (see Section 5.2), and one of the results was that the SE-RSVP daemons were relatively easy to crash. In addition, traffic aggregation for traffic between IP networks was not possible, resulting in an unnecessarily large number of reservations. These results lead to the development of a modified mechanism to establish the RSVP priority queues via a Bandwidth Broker. The Bandwidth Broker acts as a central agency to collect bandwidth requirements from multiple applications and uses ssh to directly login the RSVP routers to reconfigure the priority queues. This approach yields the following benefits:

1. Flexibility to aggregate traffic due to full control of routing queues
2. Reuse of already hardened ssh daemons, which should be harder to crash than SE-RSVP

3. Common interface on top of underlying Intserv and Diffserv technologies, which could be used transparently.

In addition to the Bandwidth Broker, we implemented a boundary controller as part of the LanContainment strategy (see Section 4), which implements an adaptive egress traffic shaping. This strategy resides on a subnet gateway, and for now it constricts the outgoing bandwidth when a flood is detected that is originating from the subnet. The LanContainment strategy uses a tool called `tc` which interacts with the Linux kernel to shape network traffic. This approach is similar to Diffserv in that it pushes the complexity to the network boundaries, and does not require routers along the path to keep internal state.

3.3 Access Control and Cryptographic Mechanisms

We needed access control at the application level to keep an attacker from gaining application-level privileges easily. In other words, without access control, the attacker can damage a critical application simply by using the application's programming interfaces directly. For example, if the application offers a “write” method, the attacker can simply write corrupt data into the application. We prevent this direct attack by:

- Requiring clients and servers to authenticate each other so that unknown clients cannot get service and unknown servers cannot provide service;
- Checking an application-wide access control policy so that unauthorized clients cannot get service and unauthorized servers cannot provide service;
- Digitally signing requests so that requests cannot be fabricated;
- Marking requests with serial numbers so that requests cannot be replayed.

Each of these four steps is part of application-level access control.

Access control should be difficult or impossible to circumvent, of course, so we require that the application use only client-server interactions for communication, and we do access control on every such interaction. Because all the APOD software is based on CORBA, this requirement was easily met.

Our implementation of application-level access control began with the Object-Oriented Domain and Type Enforcement technology (OO-DTE) from Network Associates, Inc. (NAI) [30]. This technology has three main parts:

1. An access control policy language, called DTEL++, in which one can specify which clients may have access to which services;
2. Policy distribution mechanisms, used to make global changes to the policy;
3. Access enforcement mechanisms in both clients and servers.

OO-DTE is based on CORBA, and so it seemed a good fit with the other APOD mechanisms.

When we combined OO-DTE with AQuA replication, however, we uncovered a problem. OO-DTE is based on SSL, which is a point-to-point protocol. When a client or server is replicated, though, communication has many endpoints, each of which must be handled in exactly the same way. Obviously, one might open many SSL connections, one for each replicated endpoint, but this solution does not work in the presence of AQuA. The AQuA Gateway encapsulates a multicast protocol so that opening a multicast connection to many replicas looks very similar to opening a single connection to one object. So if we opened many SSL connections we could not use AQuA and if we used AQuA, a single SSL connection could not be transported over multicast protocols.

To continue using AQuA, we abandoned OO-DTE's enforcement mechanism, i.e., part (3) listed above. Instead, we built our own enforcement mechanism and made it work with AQuA. That mechanism:

- is implemented in CORBA interceptors;
- uses the Java Cryptographic Extension (JCE) to handle public-key crypto and digital signatures;
- uses NAI's DTEL++ language unchanged.

The most difficult part of this implementation was ensuring that the state of the interceptors was transferred correctly to new multicast endpoints started up after the protocol was already in operation.

So access control in the APOD Toolkit consists of NAI's standard tools for processing the DTEL++ policy language (part 1) and a nonstandard replacement for the policy enforcement mechanism (part 3). We never got the policy distribution mechanisms (part 2) working with the other parts. Static access control is possible, then, but changing policy dynamically as part of a defense strategy was never implemented.

3.4 Intrusion Detection

Intrusion detection systems monitor computer systems and report events that may indicate security breaches. There are two basic intrusion detection technologies:

1. Signature-based and
2. Anomaly detection.

Signature-based IDSs make use of attack signatures and work best for known attacks. IDSs based on anomaly detection attempt to detect deviation from known and expected behavior that may potentially be caused by attackers. Anomaly based systems work better for novel attacks but run the risk of false positives. Both types of system run the risk of false negatives where attacks may go undetected. Signature based systems need to continuously maintain and update the database of attack signature, and wrong or improperly generalized signatures may lead to false positives as well as false negatives.

Various techniques including learning and event correlation have been used to improve the qualities of IDSs. It is not uncommon to see both types of techniques employed in a single IDS package.

Based on the parts of the system they watch over, an IDS can be categorized as either host based or network based. A network based IDS focuses on network related intrusion whereas a host based system focuses on intrusions into host machines. Table 2 provides some examples:

	Host based	Network based
Signature Based	Tripwire	Snort ²
Anomaly Detection	Emerald ³	Connection Flood Detector

Table 2 - Examples of IDS systems

Most current intrusion detection research focuses on detecting and diagnosing intrusions on hosts or networks, rather than survivability of the applications running on them. There has been effort to enable intrusion detection systems (IDSs) to interoperate [32], but for the most part, current IDSs work in isolation from other IDSs, the applications that they are protecting, and the security managers whose policies they can influence. Furthermore, many of the IDSs detect the security incident *post facto* since they are based on analysis of system logs.

Despite the shortcomings of IDSs, they constitute a valuable tool for defense. Traditionally, IDSs are deployed in the system as part of its security management plan so that a security administrator has a way to know about and react to security incidents. In defense enabling, many times the defense strategy includes a reactive part i.e., defensive adaptation to be mounted in response to some trigger. IDSs can be the source of such triggers. In addition, it is generally believed that better adaptive decision-making requires better overall awareness of the system—without the right picture of the system the defense may mount a wrong or

² Snort is primarily a signature-based mechanism, but it also makes use of some anomaly detection techniques when it attempts to detect flooding.

³ Emerald makes use of some signature matching techniques on BSM logs, but the unique strength of Emerald technology is in event correlation and detecting anomalous events.

less effective adaptive response. A sound defense strategy therefore cannot be designed without multiple mechanisms that provide awareness, and IDSs form an important class of these⁴.

We investigated several COTS and research IDSs including Emerald [33], GrIDS [34], JAM [35], Snort [36], and Tripwire [37]. We have used Snort and Tripwire as representatives of host based and network based intrusion detection systems in defense enabling. Several factors influenced our choice including cost, availability and ease of integration. Commercial IDSs are not cheap: outfitting a reasonable sized testbed could potentially strain APOD's budget. Although Tripwire is a commercial product, we have been able to use a free version. Snort is available freely. During 1999, when APOD was yet to decide on any particular IDS to experiment with, Emerald and GrIDS were not readily available. Emerald needed Solaris based hosts, and the APOD testbed consists primarily of Linux hosts. We tested JAM extensively but it was primarily a tool meant for human users, its GUI based interface made it hard to integrated as a defense mechanism and its anomaly detection capability is more effective if it is used as an offline analysis tool. For all these reasons, Snort and Tripwire were chosen to represent IDS technology in our defense enabling research.

The following are example defense strategies that rely on Intrusion Detection Systems:

Strategy 1: Network-based IDS detects attack packets originating from an IP address A . In response, block traffic from A .

There could be many variations of this simple strategy based on what "block" means and if A is the address of a machine that hosts application components. For instance, if A is a machine that does not host any application component or service, then all traffic from A can be blocked. If A hosts application components or services then specific packets from A may be blocked. Blocking may be permanent, periodic or temporary. In case of temporary blocking traffic is blocked for a certain period time. For permanent blocking the time period is infinite. In the periodic case, the blocking is removed periodically. Also the blocking could take place at each host, or at a more aggregate level such as at a firewall or a boundary controller.

Strategy 2: Host-based IDSs detect problems with a host. In response avoid that host for all critical functions.

Here also multiple variations are possible based on the host's role in the application and what avoidance means. For instance, it may be possible to block all traffic to and from the host; move application components off that host or even shut down the host.

⁴ Various probes and resource management mechanisms included in the system are the other sources of such inputs. CPU and Memory usage of hosts, available hosts in the system, available bandwidth in the links, etc., are examples of awareness inputs provided by them.

For a given IDS, more specific strategies are also possible:

Strategy 3: Copy all files critical to the application into a CD ROM back up, and monitor the files on disk using Tripwire. If Tripwire detects changes in the monitored file space, restore it from the CD.

Some of the strategies described above require the services of other defense mechanisms in addition to the IDSs. For instance, avoiding a suspected host may involve a replication management mechanism and blocking traffic will involve a packet filtering mechanism.

One of the advantages of using the IDSs is that they provide early warnings. A malicious attacker intending to mount a damaging attack on a system will need to gather information about the system first. The type of application APOD intends to defend are those for which information is not readily available, i.e., the attacker will have to do surveillance on the system first. If the attacker is an outsider, after gathering information about the system the attacker will need to obtain a foothold in the system by taking over or corrupting a host that is part of the application. The strategies described in this section can be thought of as early intervention during these stages of the attack. The defensive responses mounted as part of these strategies may disrupt the attack and in some cases may stop it, but in either case the outcome is not guaranteed.

IDSs are known to be imperfect: they may raise an alarm when there is no real threat and they may fail to raise an alarm when there is a real attack. Furthermore, there may be inherent limitations of individual IDSs. For instance, Snort can be fooled by source-address spoofing. Tripwire based strategies cannot be applied to transient and data files that are legitimately created and altered by the application.

APOD attempts to mitigate these risks within the defense strategy. For instance, false positives may lead to unnecessary adaptations. The defense may observe the system after the adaptive response is engaged and roll back the adaptation if it decides that the adaptation was unnecessary. To address the limitation of IDSs the defense strategy should not rely solely on IDSs. Multiple IDSs can be used to improve coverage, correlation among detected events and IDS that do correlation can be used improve accuracy of detection.

Finally, there are opportunities to improve both the defense and intrusion detection by feeding intelligence and awareness from other defense mechanisms to IDSs and vice versa. As an example, consider a variation of the strategy where replication management and IDS are used as defense mechanisms. If an IDS suspects that a host was being port-scanned and the replication management mechanism detects that replicas are dying on that host, then combining these alerts will give a stronger indication that the suspected host is under attack and a stronger justification for moving replicas away from that host.

3.5 Firewalls

3.5.1 Motivation

Consider a network consisting of a set of LANs that are interconnected via a WAN. In order to fulfill security requirements, security engineers traditionally focus on the low-level design and implementation of the network as a means of securing each LAN. Limiting Ethernet broadcast domains via switches and IP subnetting are examples of network security design decisions. In addition, firewalls placed at the boundaries between LANs and WANs can be used to protect LANs from outside attacks.

The main functions of a firewall are:

- Traffic shaping and filtering
- Network Address Translation (NAT)
- Virtual Private Networking (VPN)
- Application level access control

APOD uses firewalls primarily for the first of these, so traffic shaping and filtering will be the primary topic of this section. See Sections 3.6, 3.7, and 3.3 for the APOD approach to NAT, VPN, and access control, respectively.

The exact configuration of a firewall is a very sensitive piece of information, since it could give attackers a chance to find holes in the defense. In addition, changing a single configuration parameter on a firewall can easily separate the LAN off from the WAN, thereby causing massive outages. For these reasons, security engineers tend to think of a firewall configuration as something to be figured out carefully once, and locked in place ever after.

The problem with static use of firewalls is its all-or-nothing nature: As long as attackers are unable to penetrate the firewall, a static configuration makes sense. However, as soon as attackers find a hole in the configuration, they have access to the protected LAN until the system administrators manually change the firewall configuration, which usually takes quite a long time.

In APOD, we have investigated multiple ways of automatically changing firewall configurations in an attempt to contain the attacker. We've used firewalls in the following two ways:

Host-based Firewalls

In a host firewall approach, lightweight firewalls are deployed at the very edges of the network, namely the hosts themselves [2]. This enables finer grained traffic filtering and policy enforcement. In addition, it limits misconfiguration effects to the end system itself.

APOD uses COTS intrusion detections systems like Snort and Tripwire to detect attacks originating from specific hosts. Once an attack is detected, firewall rules can be changed to:

1. Contain the attacker on the host by blocking all outgoing traffic on that host
2. Limit attacks by blocking all incoming traffic from that host on other "non-infected" hosts

See Section 4 for a more detailed description of this defense strategy.

Network-based Firewalls

In a network firewall approach, more complex firewalls are deployed at the boundaries between LANs and other networks. Network firewalls are traditionally used to protect LANs from outside attacks.

In addition to the COTS systems used in the host-based scenario, APOD monitors outgoing traffic and upon detecting an outgoing flood, adaptive rate-limiting is enabled in the firewall to prevent the LANs from being used as launching pads for insider flooding attacks. Since the host-based approach cannot contain attackers that operate on hosts that are outside of APOD's control, we've augmented network firewalls with an adaptive trace back capability to block those attackers closest to their source.

3.5.2 Design and Implementation

The following points summarize the use of firewalls in the defensive policies described in chapter 3.

Blocking of attack traffic

Based on detecting the origin of an attack to be on a certain host, APOD enabled hosts will block all traffic from and to that host. We have used the Linux-based `iptables` firewall for this purpose [4]

Rat-limiting of floods

Based on detecting an outgoing flood in an APOD boundary controller, the boundary controller will rate-limit outgoing traffic using a token bucket filter model for a certain time interval. Rate-limiting is implemented using the Linux `iproute2` package [5], [6]

3.5.3 Related Work

Besides using open-source Linux firewall utilities such as `iptables`, we've also investigated how to integrate APOD with the following COTS firewall products.

Use of Xtradyne's Domain Boundary Controller as a Network Firewall [1]

The Xtradyne Domain Boundary Controller (DBC) is a CORBA Application Level Firewall that securely transmits CORBA requests and replies across the domain boundaries including packet filter firewalls and NAT Routers. Acting as a CORBA Firewall the DBC checks the correctness of IIOP messages (or RMI/IIOP messages respectively) and filters out hostile and destructive messages.

The DBC can be chained in with the already existing network firewall that APOD uses to provide CORBA level filtering capabilities. The relatively high memory and CPU consumption of the DBC, however, make it difficult to deploy as host firewalls.

Firewall on a NIC [2]

The architecture of this system consists of embedded host firewalls that are centrally managed from a policy server. Integration points with APOD could be:

1. Embed distributed APOD components onto the same NIC in order to implement dynamic firewall reconfiguration policy
2. Establish an interface between APOD and the policy server to have APOD reconfigure the NICs via the policy server.

Cisco IOS Firewall [8]

Interfacing APOD with Cisco IOS Firewalls via `ssh` seems to be a viable solution to transfer APOD technologies into deployed networks.

3.6 Port/Address Hopping

3.6.1 Motivation

When a client communicates with a server over TCP/IP, all packets exchanged between the two parties contain a source <address:port> and a target <address:port> pair. This is sensitive information, and attackers spend time sniffing network traffic in order to get addresses of potential targets. Traditional security mechanisms try to thwart attackers by encrypting and encapsulating data packets using VPN technologies like IPsec. However, after encrypting the payload information and encapsulating the packet, the encrypted data is still sent between two dedicated endpoints. This leaves attackers the possibility of detecting endpoints via traffic analysis and running attacks against the endpoints.

Port and Address Hopping is an attempt to obfuscate the “real” ports and addresses by replacing them with values picked randomly from a range of possible values. Packets intercepted by attackers will only reveal the random addresses, which are valid only for a limited time period, i.e. 5 minutes. In order for attacks against a server to be successful, the attacker must discover the current ports and execute an attack using them all within the port-refresh period. Furthermore, an attempt to use ports that are different from the one picked randomly by the hopping mechanism, or that were used in previous cycles, will raise a warning alarm.

3.6.2 Algorithm

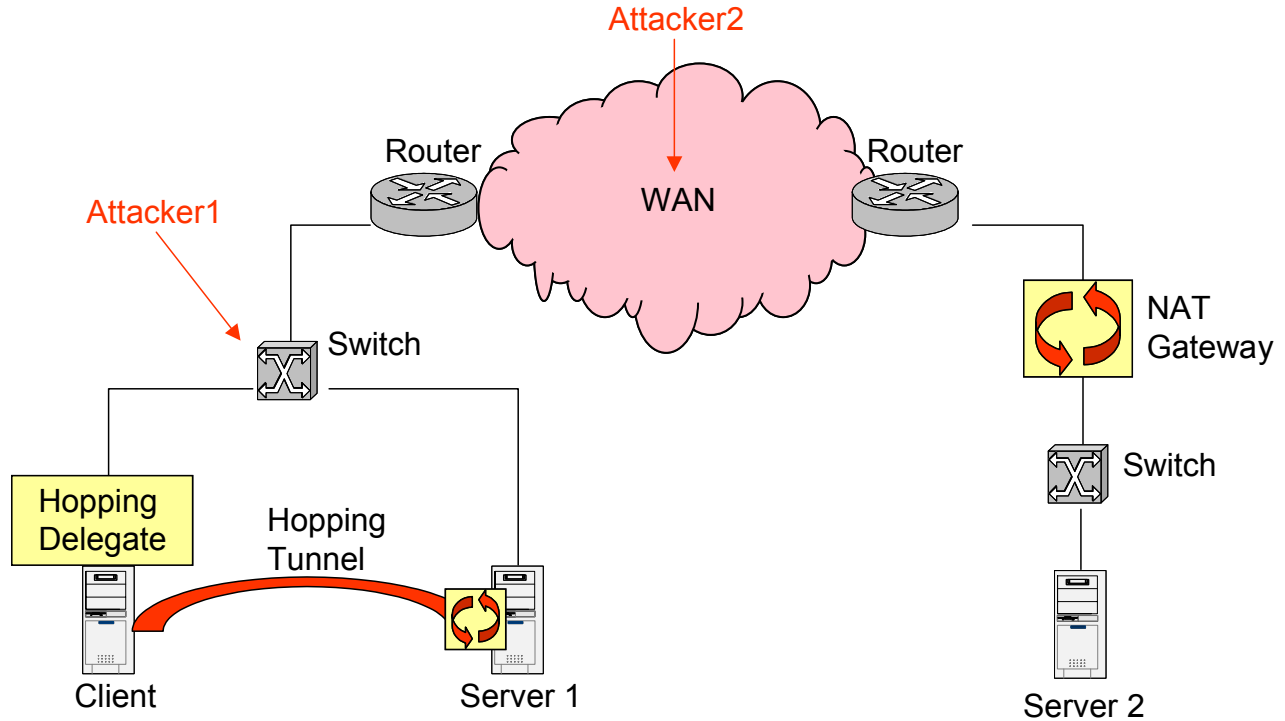


Figure 2 - Address / Port Hopping via Tunnels and/or NAT Gateways

The hopping mechanism is implemented by a client component and a gateway. The client component is located on the client itself. It intercepts all communication to the real server, and replaces its $\langle \text{realaddress}:\text{realport} \rangle$ pair by $\langle \text{fakeaddress}:\text{fakeport} \rangle$. The gateway component is located either on the server's LAN or the server itself. It does the reverse mapping from $\langle \text{fakeaddress}:\text{fakeport} \rangle$ to $\langle \text{realaddress}:\text{realport} \rangle$. The $\langle \text{fakeaddress}:\text{fakeport} \rangle$ pair is picked randomly from a range of IP addresses and ports.

The address/port pair is used for a specific cycle time, after which a new pair is generated and used. Information about the previous pair is saved, in order to be able to identify suspicious traffic using later. Note that this mechanism relies on synchronization of the random number generators used in the two components, which can be achieved by seeding both generators with the same value. In addition, time synchronization is required to synchronize the switchover to using a newly generated $\langle \text{fakeaddress}:\text{fakeport} \rangle$ pair.

3.6.3 Design and Implementation

We've identified the following use cases, which lead to two different designs and implementations (see Figure 2):

Tunnels - client and server have addresses within the same IP network

The communication between Client and Server1 does not involve any routers, since both machines are on the same IP network and connected via a Switch. In order to prevent Attacker1 from detecting Server1's port, the Client's delegate redirects traffic to be forwarded through a tunnel to Server1, changing server side ports (and also possibly client side ports) of the tunnel randomly over time. In this scheme, our adaptive defense is limited to port hopping.

NAT Gateway - client and server are on different IP networks

For communication between Client and Server2, the hopping delegate changes the target address of packets destined for Server2 to a random IP address within the same IP network as Server2. In addition, a random port is selected. This packet is routed to the NAT gateway, which forwards the packet to Server2. Replies from Server2 to Client are again forwarded through the NAT gateway to have their source IP address and port adjusted to the random selection. Both Attacker1 and Attacker2 see only packets between Client and random IP addresses & random ports. If required, deployment of an equivalent NAT gateway on the client's LAN would obfuscate the client's IP address and port: Attacker2 would only see traffic between random addresses:ports of hosts in the two LANs.

3.6.4 Related Work

The DYNAT [3] project has implemented and validated a similar approach as the NAT Gateway described in the previous paragraph. However, the APOD solution relies on standard COTS utilities, such as Linux iptables [4] and zebecde tunnels [9], to implement the desired functionality, while DYNAT is a more hardware integrated, specialized solution.

The CONTRA project [16] implements IP address dispersion by adding CONTRA headers to packets, with the header containing the real destination. The packet addresses are then transformed and relayed over a set

of relay hosts to the final target. The relay operation includes decrypting the CONTRA header, extracting the real destination, changing the padding, reencrypting with the key of the next hop. Compared to the DYNAT approach or APOD's address hopping approach, CONTRA requires changes to the routing infrastructure (i.e., relay hosts) to support the CONTRA protocol.

3.7 Virtual Private Networks

3.7.1 Motivation

Distributed applications need to be able to communicate information between nodes in a secure way. However, most applications are not developed with security in mind, which leaves us with the difficult task of adding security features on top of already existing applications. To solve this problem, people have turned to network based security solutions, or VPN technologies, for services such as encryption and authentication. Most VPN solutions, like IPsec, operate between ISO layer 3 (IP) and 4 (TCP), which avoids interfacing with application protocols at the higher layers, and therefore enable transparent deployment of security features.

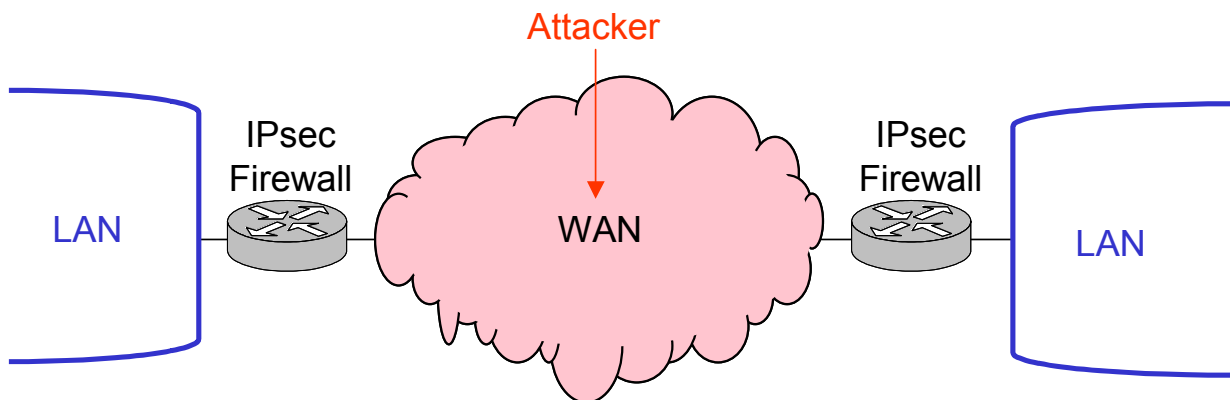


Figure 3 - Use of IPsec to provide WAN VPN functionality

3.7.2 Design and Implementation

APOD has been integrated with IPsec (see Figure 3) and encrypted tunnels. Although both of these mechanisms provide VPN functionality, they are representatives of two different categories with different strengths and weaknesses.

Heavy but powerful - IPsec [10,11]

Software integrated with APOD:

- FreeS/WAN [10]

Pros:

- Powerful - IPsec allows specifying in great detail how to encrypt and/or authenticate packets. Multiple configurations can be used, especially a mode in which only authentication is deployed. Furthermore, IPsec can be configured in two modes: tunneling and transport mode. In addition, IPsec defines a key exchange protocol (IKE) to facilitate dynamic rekeying.
- Industry Standard - IPsec is defined in great detail in a whole set of RFCs [11,12]. This enables vendor independent deployment of IPsec as a network based VPN technology.
- Hardware Integration - IPsec capabilities are frequently integrated into Firewalls to provide WAN VPNs.

Cons:

- Complexity - Due to its various configuration options, it is non-trivial to setup and maintain IPsec. This confines its use to central firewalls.
- Kernel Space - FreeS/WAN, the Linux IPsec implementation, is tightly integrated into the Linux routing facility. Therefore, changing the IPsec configuration requires root privileges on the system.

Lightweight and flexible - User Tunneling [9]

Software integrated with APOD:

- SSH [13]
- Zebedee [9]

Pros:

- Flexibility - Tunnels can be used to establish VPNs between two dedicated end-systems or between two networks.
- Dynamic Nature - Most user tunnel programs are very lightweight, which facilitates dynamic behavior
- Easy Integration with Port hopping and OO-DTE - Because of its flexibility and dynamic nature, user

tunnels make an excellent choice for port hopping between end systems, especially if both end systems are on the same IP network. See the chapter about port/address hopping for more details.

- User Space - User tunnels can be established by regular users, which eliminates the need for root permissions.

Cons:

- Non-Standard - With the exception of SSH, many user tunneling programs are non-standard, which makes cross-platform compatibility difficult.

3.7.3 Related Work

Many IPsec implementations are deployed and in use. We are currently investigating how to enable Cisco's IPsec implementation [14] and use it together with APOD. In addition, IPsec is also deployed in VPNshield [15]. Dynamic behavior is achieved by combining IPsec with intrusion detection systems and RSVP.

3.8 *TCP Connection Flood Defense Mechanism*

3.8.1 Motivation

Normal TCP/IP networking is open to an attack known as "TCP connection flooding". This denial-of-service attack prevents legitimate remote users from being able to connect to your computer during an ongoing attack and requires very little work from the attacker, who can operate from anywhere on the Internet.

TCP connection floods are situations in which attackers establish a large number of connections to essential server ports, and hold on to the connection as long as possible. This results in resource exhaustion of server side sockets, which will leave the server in a state where legitimate clients cannot establish connections to the server anymore.

TCP connection flooding is different from SYN Floods, in that SYN floods only send massive amounts of SYN packets, without the intent of establishing connections. IP spoofing can therefore be easily combined with SYN floods, but it is more difficult to spoof connection floods, since replies have to be able to get back to the source IP address to complete the three-way SYN-ACK-SYN/ACK handshake.

Luckily, SYN cookies [18,56] provide COTS protection against SYN floods. If deployed, the TCP/IP stack will use a cryptographic challenge protocol known as "SYN cookies" to enable legitimate users to continue to connect, even when your machine is under attack. There is no need for the legitimate users to change their TCP/IP software; SYN cookies work transparently to them.

APOD has developed a reusable defense mechanism against TCP connection flooding, which will be detailed in the next section.

3.8.2 Design and Implementation

The APOD defense against TCP connection flooding is implemented as a rapid reaction loop (Figure 4). For a given host and a set of ports, APOD monitors the number of connections established to the ports. If this number exceeds a certain threshold, all traffic to and from the source of the violation host will be dropped using a host-based firewall like `iptables` [4]. The potential danger of blocking spoofed source addresses is limited by the fact that the source IP address has to establish and maintain a live TCP connection.

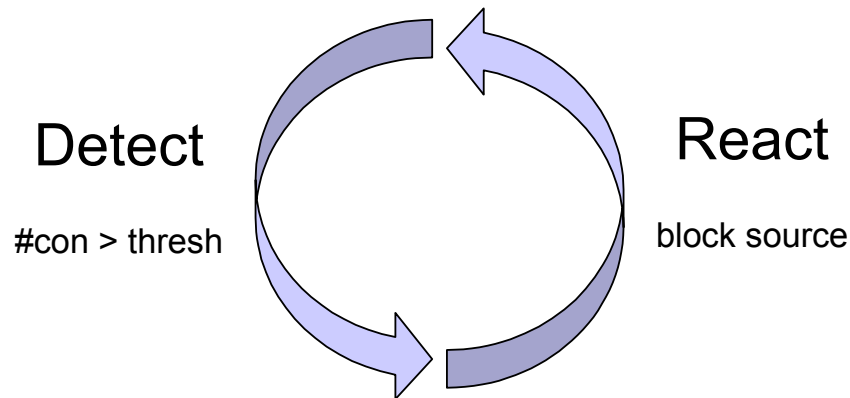


Figure 4 - TCP Connection Flooding Defense

3.8.3 Related Work

The tunneling program Zebedee [9] has been hardened to deal with DDOS attacks. It allows users to rate-limit the number of established connections per second. In addition, idle TCP connections can be closed down after a timeout. A feature called "readtimeout" tries to deal with connection flooding by specifying a

time limit in seconds within which reads from a Zebedee tunnel must be complete. However, applying a timeout will have some impact on performance and an unnecessarily small timeout may cause valid connections to fail.

3.9 *ARP Spoof Detection*

3.9.1 Motivation

ARP (Address Resolution Protocol) spoofing, also known as ARP cache poisoning, is often used to intercept traffic on a local subnet. The attack involves broadcasting invalid ARP messages. There are tools available on the Internet that allow one to easily create an ARP message with IP/Ethernet address mappings as specified by the user. This allows the attacker to create ARP messages that force one or more IP addresses to map to the attacker's own Ethernet address, causing all traffic bound for those IP addresses to arrive at the attacker's Ethernet address instead. The host to which the traffic was originally sent will ignore the traffic because the Ethernet addresses in the traffic header do not match its own. The attacker's intercepting host can either forward the intercepted traffic or simply keep it; in the latter case, the original destination host will become effectively isolated.

During the APOD Red Team experiments, the Red Team used ARP spoofing in its attacks. In response, the APOD team developed a mechanism for detecting the ARP cache update messages the attacker uses to poison the ARP cache.

3.9.2 Design and Implementation

APOD's ArpCache Spoof Detection mechanism caches IP/Ethernet address pairs and continually checks for changes in the pairs. If an attacker broadcasts an invalid pair, then the mechanism will see this pair and raise an alert. This strategy works fine in a subnet that does not use dynamic IP address assignment.

3.9.3 Related Work

There are other tools that can help monitor and detect ARP spoofing. One such tool, called ArpWatch, monitors IP/Ethernet mappings for changes and notifies an administrator. Another possible defense is to use static ARP mappings. ARP update messages from the network are ignored if a host is configured with static ARP mappings and therefore an attacker is unable to spoof at the Ethernet layer.

3.10 Host Shutdown

3.10.1 Motivation

No matter how hard a defensive system tries to keep the attacker out, there is always the possibility that the attacker will eventually take over one of the systems and use it as a launching point for further attacks.

In order to prevent infiltrated systems from being used as launching pads, APOD falls back to halting a host on which intrusions have been detected. However, blindly shutting down hosts only based on the information from IDSs would leave the system vulnerable to false positives and DDOS attacks triggered by attackers who just trigger the IDS without actually causing an intrusion. Therefore, shutting down hosts must be done in a coordinated way that is based on more global knowledge. Section 4 talks about how host shutdown is used in a particular defense strategy.

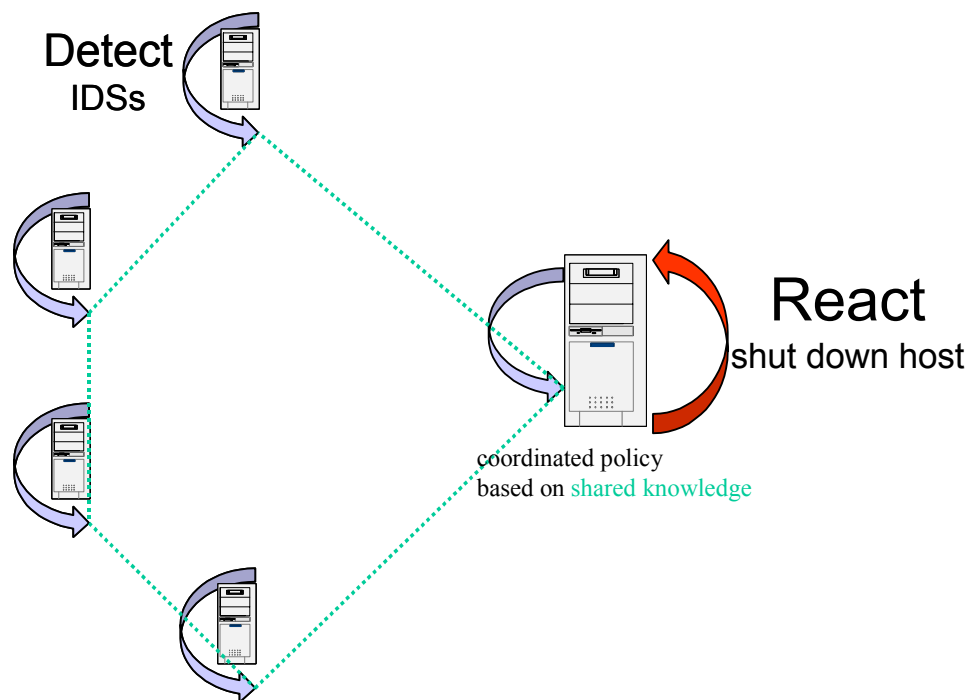


Figure 5 - Host Shutdown triggered by coordinated policy

3.10.2 Design and Implementation

The host shutdown mechanism itself is as trivial as calling "halt" on a Linux system. However, most of the design complexity went into designing the right policy that only shuts down a system when this is the right thing to do. Read chapter 4 for more information about how host shutdown is used in a coordinated defense policy. Figure 5 displays the overall architecture for executing a coordinated shutdown.

3.11 QuO Adaptive Middleware

QuO (which stands for Quality Objects) is a middleware framework, developed by BBN, for building applications that are aware of their environment and can adapt to changes in it. QuO applications can specify their non-functional requirements (e.g., security, performance, or dependability requirements), measure what is being provided, access interfaces for controlling the desired level of service, and adapt to changes in levels of service. While this research developing QuO was originally performed in the areas of network quality of service and open implementation, it has been used extensively in defense enabling because of its support for adaptation and its capability to integrate various mechanisms. By building the defense in middleware it is also possible to separate the specification and implementation of the defense from the functional aspects of an application. This facilitates reuse of parts of the defense across multiple applications as well.

The QuO adaptive middleware framework enables the following:

- *The development of intrusion- and security-aware applications:* IDS and security tools usually do not interact with the applications that run on the infrastructure they protect. Using the QuO middleware it is possible to make the applications aware of the system level events and anomalies and the security and IDS mechanisms aware of application level events and anomalies.
- *Integration of resource managers, IDS and security tools as defense mechanisms:* Non-trivial defense strategies use the capabilities of multiple mechanisms that are not directly involved with the functional aspects of the application. QuO middleware is used to integrate these mechanisms with one another and with the application being defended. A special case of this kind is the *integration and interfacing of multiple IDSs at the application level*, where the benefit of having different IDSs working in cooperation can be further extended by an application level perspective.
- *The development of survivable applications:* Adaptation is fundamental for survival. Cyber attacks result in changes in the operating environment of an application such as consumption or corruption of resources. A survivable application must either adapt itself to cope with the changes or initiate reconfiguration of the environment to compensate and recover from the changes. With the advent of advanced middleware such as QuO that provides architectural support for this kind

of adaptation and a conceptual separation from the functional aspect of the application, systematic development of survivable applications is now possible.

Details about the QuO middleware and its application in defense enabling is published in various papers [38], [39] including the ones that came out of APOD research [40], [41]. In this section we will provide a brief summary.

The QuO middleware framework provides a set of high-level languages known as QDLs (for Quality Description Language). QDLs take the same level of abstraction as Interface Description Languages in distributed object computing paradigm. They allow systematic specification of:

- An application's Quality of Service (QoS) requirements and
- Its adaptive response when these requirements are not met.

Although designed initially to capture traditional application level QoS aspects such as response time, number of frames in a video transmission, etc., QDL can be used to capture any non-functional requirement that is important for the application. In the present context, we have used QDL to address survivability requirements of the application.

QuO code generators and its runtime kernel help to integrate the QoS-based adaptation into the application. The code generators translate the QDL specification into the programming language constructs such as Java or C++ classes, which then can be compiled with the application code implementing the functional aspects of the application. The change needed in the application code to integrate the QoS and adaptive aspects are minimal.

The following two figures (6 & 7) explain how the code produced by the QuO code generators is integrated in the overall application.

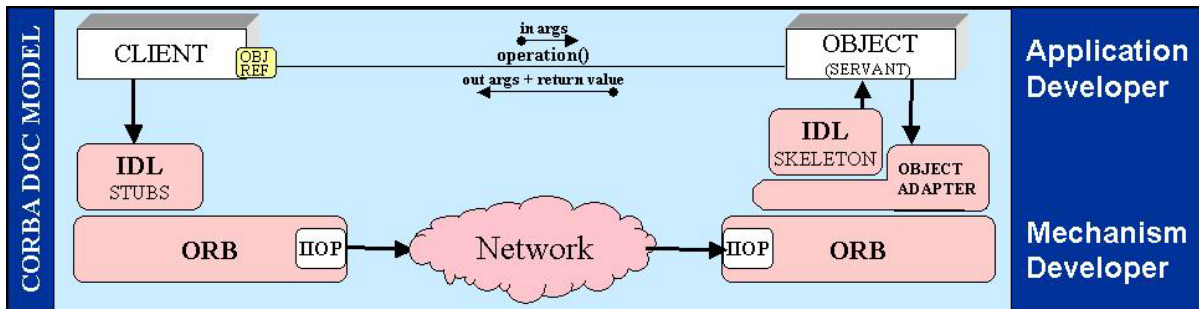


Figure 6 - Inter object interaction in Distributed Objects Computing (DOC)

Figure 6 shows a typical interaction between two distributed objects. The client makes a call to a stub as if the call is local. The underlying mechanism then makes the actual remote call, obtains the results and presents the result back to the caller. The application developer need not worry about the remote call. In the QuO framework, this simple call path is slightly modified by the insertion of QuO components.

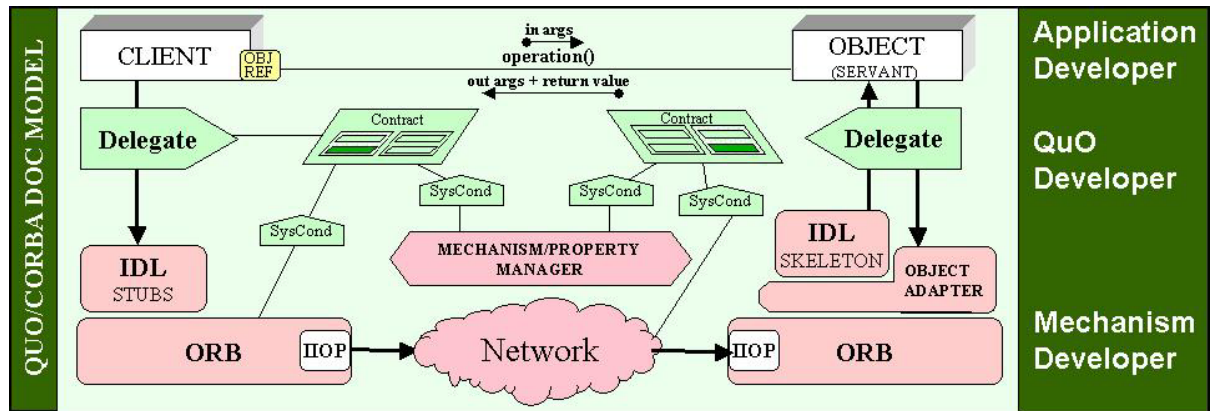


Figure 7 - Interposition of QuO Code Generated components in the DOC inter-object interaction

As shown in Figure 7, the client's call is intercepted by a QuO component, the delegate, before it gets to the stub. The delegate is interconnected with two other kinds of QuO component: contracts and system condition objects, or SysConds. After intercepting the call, the delegate makes use of the QuO runtime to evaluate the contract. As part of the QDL specification, the contract essentially characterizes the QoS regions in the operating space of the application. Contract evaluation results in determining the region in that space the application is currently in. In doing this contract evaluation, the QuO runtime makes use of various system conditions that project present state of various variables used in defining the operating space. In addition to projecting environmental conditions into the QuO runtime, the system condition objects can be used to send control signals to adapt and reconfigure the environment as well. The QuO framework comes with a set of system conditions that could be readily used with the QuO runtime and also provides a foundation for creating custom system condition. The delegate, which is an instance of a class generated from QDL specs, decides how to handle the call based on the result of contract evaluation: this is where QuO interjects QoS oriented adaptation. Using QDL it is possible to define handling of inter-object interaction based on the current QoS state as defined by the contract regions. Contract and adaptive behavior specification and any custom system condition required to support the contract constitute the additional development to introduce QoS adaptation to the application. We envision this development to be done separately from the application or the mechanism, by QoS-savvy middleware specialists (denoted in Figure 7 as QuO Developers). In the present context, these specialists should be survivability experts, and

the contracts will encapsulate specific survivability strategies and system conditions to interface with defense mechanisms.

As an example of how QuO contracts can be used to support survivability consider the following simple example, presented in an abstract manner i.e., not in QDL. Let us assume a simple application where a server offers a service to a set of clients. One can consider the operating space of this application to consist of two regions: “no-attack” and “under-attack”. When there is no reason to believe or suspect that the server is under attack, the application’s operating region is “no-attack”. If there is some indication that the server is under attack, the application’s operating region will be changed to “under-attack”. The behavior of the application, primarily the way the client gets the service it needs, may adapt depending on the current operating region. For example, one kind of adaptation may be to stop using the suspected server. Depending on the application’s survivability requirements, there are various ways to define the operating regions and the adaptive behavior. For instance, inputs from IDSs monitoring the server platform may be used to define the operating regions: if an IDS raises an alarm, then the operating region is “under-attack”. Otherwise the operating region is “no-attack”. The IDSs will interface with QuO system conditions to project their state (alarm or its absence) to the QuO contract. As part of the survivability strategy, one may start two independent servers at two different sites. One site is to be used as the primary and handles the service requests most of the time. This site has normal level of security protection and is built using COTS products. The other site is to be used as a back up and is protected by higher level of security. This site uses custom and special purpose components to achieve this higher level of security and thereby more expensive to use all the time. Given this, the under-attack and no-attack regions are with respect to the primary server. When the application is in “under-attack”, client requests will be redirected to the back up server. This behavior will be specified in QDL and will become part of the application’s runtime code via the QuO code generators. Table 3 represents the operating regions and associated behavior corresponding to the contract described here.

	No-attack	Under-attack
Definition	Absence of IDS Alerts	Presence of IDS Alerts
Behavior	Pass through client request to Primary	Redirect client request to Secondary

Table 3 - Operating Regions of a sample Defense Contract

This simplistic example can be extended towards a more realistic situation. One can think of having multiple IDSs monitoring the primary server platform. The operating space may have more regions with various levels of suspicion as opposed to the two basic “under-attack” and “no-attack” regions. Behavioral adaptation may include temporarily blocking the client’s request, using cached values to service the clients or replacing the suspected server.

As middleware, QuO enables integration of various mechanisms. Over the course of the APOD project, we needed to integrate various defense mechanisms providing essential services for particular defense strategies. This led us to an architectural pattern illustrated in Figure 8.

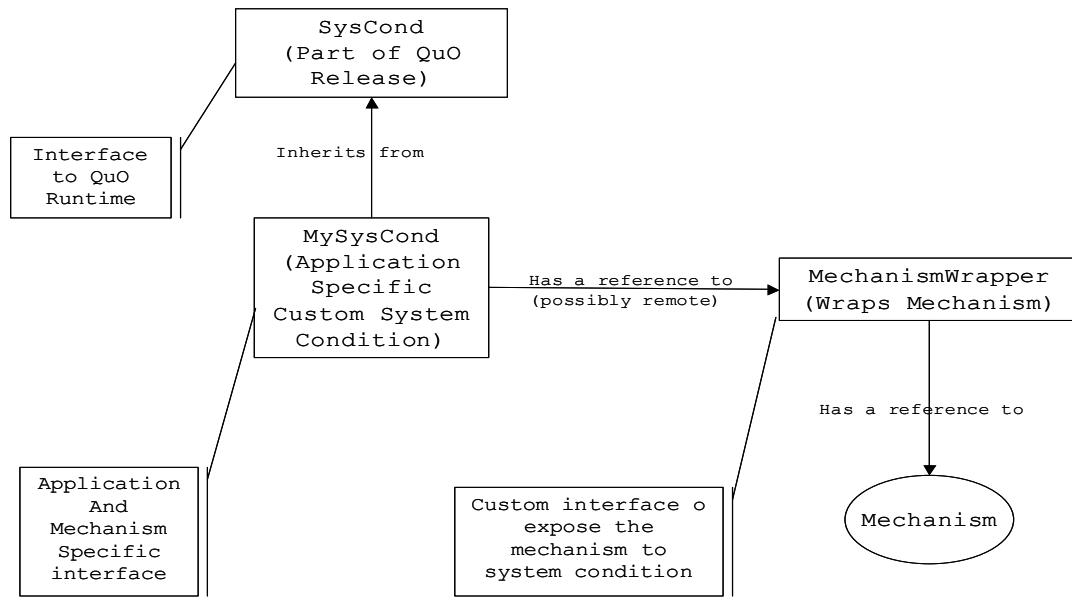


Figure 8 - Pattern for integrating a mechanism

Figure 8 shows how a mechanism is integrated with a system condition. In this context, the mechanism can provide input to contract evaluation as well as being able to execute commands that are invoked as part of contract region changes (known as transitions). Since multiple system conditions can be used in a single QuO contract, it is possible to integrate multiple mechanisms in the context of the operating space defined by a single contract. In addition, multiple system conditions in different contracts can share the same mechanism instance via the mechanism wrapper interface. But such integration is not always simple. The next section reports some of the issues we encountered.

3.12 Mechanism Integration

Various issues arise when using the integration pattern described in the previous section. First and foremost is the issue of interface. Most often the mechanisms are designed to operate standalone and offer a textual or graphical interface for a human user. For the type of integration we need, an API (application programming interface) is needed through which QuO can send commands to and receive information from the mechanism. Depending on the particular mechanism at hand, it may be possible to create a wrapper

around the mechanism to create the desired API, but we have encountered cases where it is not possible to do that in a simple or non-intrusive manner. A case in point is the JAM intrusion detection system. This anomaly detection based IDS showed promise in recognizing novel attacks, but it only had a GUI and did not offer any API or hooks to create a wrapper around it. On the other hand, it was possible to create a wrapper around the Snort IDS so that QuO could activate it and receive alerts from it.

Second, some mechanisms are designed to be run offline without any direct interaction with the application we want to protect. For instance, Tripwire is meant to be run by system administrators on a regular schedule, but having humans in the loop is not appropriate for the largely automated defense we wanted. If the attacker corrupts files that contain critical information for the application (e.g., static tables, executables, dynamic libraries), it would be better for the defense to become aware of the corruption as soon as possible. Besides, monitoring key files may also be important for catching the intruder while or before he tries to erase his footprints. For some user-driven mechanisms (Tripwire is one) it is possible to create an automated closed loop, from which logs are analyzed as soon as they are generated, but for some others it was not possible (for instance JAM).

Third, when several mechanisms are integrated, there may be conflicting assumptions or requirements. One important example for APOD was integration of active replication with application-level access control. The access control required that inter-object communication be intercepted and its authorization checked. Active replication required that multiple object replicas handle this access control in identical ways. In principal, these requirements do not conflict, but in practice, their implementations may. In the case of OO-DTE access control and AQuA replication, the access control mechanism prevented the replication mechanism from working. More detail on this conflict and our resolution of it can be found in Section 3.3.

Although we encountered these issues in the context of defense enabling in APOD, integration issues are natural to distributed middleware research. In APOD our goal was to demonstrate the value of integrating defense mechanisms, so many of the integration was done manually and in an ad-hoc manner. We are now investigating how such integration can be done in systematic way and what kind of language and tools support can be developed.

4 Defense Strategy

4.1 Local vs. Global Adaptation

While adaptation is essential for survival, it also raises problems. First, inaccurate information may lead to unnecessary and costly adaptation. Second, an adaptation mechanism could be abused by the attacker to

cause denial of service or to push the application into a less desirable situation. Therefore, one needs to be careful in designing the defense strategy that controls adaptation.

We claim that a flexible defense needs to have a range of adaptive responses, some of which are taken as a quick reaction to possible attack and some others requiring deliberation and coordination among multiple components before any action is taken. Some adaptations are local in scope and some involve a large part of the system. Some affect only a single host (for instance, recovering files on disk), while others affect network activities (for instance, blocking traffic from a source address) and still others affect the application (for instance, delaying a request, or using an alternate service provider). Some responses are lightweight in the sense that they can be reversed fairly easily (for example, an individual host blocking traffic from a source) and quickly, and others are not (for example, shutting down a subnet). Responses that are lightweight and local in scope are better suited for triggering quickly when information about the attack is still uncertain.

A good defense strategy needs to pay attention to all these aspects and it usually helps to have a structured way to define various levels of locality. For instance, at one level an adaptive response that affects an individual host may be local, but at another level all hosts in a network segment guarded by a router or firewall may be local (for example, rate-limiting the aggregate outgoing traffic through this router). Furthermore, it is dangerous to have a centralized controller for all kinds of adaptation for this controller would be a single point of failure, and another potential candidate for attacker abuse. Even with decentralization, it is important that the power of one controller is restricted to its own locality to prevent an attacker abusing a corrupt controller to affect other parts of the system. While considering the effect of adaptation, one needs to look beyond the physical structure. For instance, if the adaptive response is to shut down a host, physically it is affecting a single host, but its effect may be far reaching. Adaptive responses that have a broad impact are good candidates for abuse by attackers. More coordination is needed for adaptive responses that span a broader structure or have a wider impact. For example, it would be dangerous to shutdown a host based on one observer's decision, whereas it may be okay if the observer stops using the suspect host temporarily.

Using the QuO middleware for packaging adaptation control (known as Qoskets [57]), we have developed and experimented with various flavors of coordinated responses. These are described later in this section.

4.2 A Generic Strategy

APOD defense strategies are implemented using the QuO qosket component [57]. A qosket is defined independently of any application objects and represents a bundle of specifications and implementations of

an adaptive behavior. In the case of APOD, a qosket contains a self-contained adaptive defense policy. During the process of Red Team testing of APOD, we developed two defense policies.

4.2.1 Outrunning

In order to survive, a critical application must be able to move away from suspected hosts. The outrun strategy's purpose is to move 'replicated' pieces of an application away from an intruder or an infiltrated host. This allows the application to move to a more reliable host that should hopefully provide a clean environment for our application processes. The outrun strategy is implemented in the `bus_selecthost` qosket and uses APOD's software bus to accomplish the replication of application objects.

4.2.2 Containment

The second strategy we developed for the Red Team experiments was quarantining of hosts that are suspected as infiltrated by an attacker. To quarantine a host H means to have all hosts participating in the defense-enabled application block all communication to and from H. A quarantined host would be of little value to the attacker since it would be unable to communicate with other hosts.

The containment strategy implements quarantining of hosts. APOD differentiates hosts by whether or not they are used by the APOD-enabled application. An application host will detect bad remote hosts using a network intrusion detection system (Snort) and detect itself as bad using a file integrity system (Tripwire). The containment is done using two mechanisms, a firewall and shutdown command. The firewall can block all traffic to and from specified hosts. The shutdown ability is available on application hosts and comes with two choices: (1) shut down the application processes on the host or (2) shut down the whole host. The second option should be the more common choice. It will totally remove the host as a possible resource for the attacker.

4.2.3 Flood prevention and trace back

In addition to using the defense strategies above, a new strategy named "Flood prevention and trace back" was developed to deal with network based flooding attacks. It was deployed via a set of pre-established RSVP reservations using SE-RSVP together with coordinated defense behavior bundled into the LAN containment qosket. The purpose of the RSVP reservations was simply to prioritize application traffic and traffic from the APOD components, which allows the system to tolerate network floods. In addition, each

LAN was guarded by a boundary controller running an instance of the LAN containment qosket. Upon detecting a significant increase in outgoing traffic, the boundary controller starts rate-limiting that traffic for a certain amount of time. In addition, it hooks into the already existing ring of containment qoskets, and will block a host closest to its source if the host is located on the LAN it controls.

4.3 Defense Policy Options

The previous section's strategies have many interesting policy issues. Some of these issues were debated during the Red Team testing of APOD.

4.3.1 Outrunning

In the outrun strategy two major policy issues were discussed. One issue is the placement of new replicas and the other issue is the tradeoff between the number of replicas and application throughput. The most basic placement scheme is to put new replicas on hosts that have no replicas. This is the default behavior for the APOD software bus. The outrun strategy goes further by selecting not only a host with no other replicas but also a different host and different LAN, if possible, than the previous replica's host. Also, input from the containment strategy about contaminated hosts is used to help determine a clean host. The question of the number of replicas was discussed but only strategies with a fixed number of replicas were implemented. We speculate that increasing the number of replicas while under attack would increase the survival time of the application more than it would decrease the application's throughput due to replication overhead.

4.3.2 Containment

The act of quarantining a host is a drastic action. Currently APOD quarantines a host if either a local sensor is triggered on that host or a network sensor is triggered on another host. In the containment qosket, if the network sensor detects malicious network traffic from host A, then the qosket will trigger a quarantine of host A. The problem is that attackers can easily spoof communication to look like it came from a different host. At one extreme, an attacker might be able to trick the containment strategy into quarantining all the application hosts and thus kill the application. At the other extreme, if a containment strategy were to depend only on the local host detection mechanisms, then an attacker who could disable the local detectors quickly would never be quarantined. Some solutions we have discussed, between these two extremes, count the number of IDS alarms and remember whether they are from different sources. We stratify the alarms, giving a combination of network detection, file system detection, and local death of a

replica as the highest mark for a host being bad. Different combinations of detections define the lower marks until the least mark, which contained only network detection as a source. Other ideas about repeated warnings and source of warning have been contemplated. The original design of the containment qosket required every containment qosket to agree with the quarantine before it was implemented.

4.3.3 Flood prevention and trace back

LAN containment is even more drastic than host shutdown. Boundary controllers easily become the prime target for attacker, since causing disruption on the boundary controller escalates into affecting all machines in that LAN. Automatic actions on the boundary controller were therefore mainly limited to doing adaptive egress filtering to prevent outgoing floods. To get a good estimate of the expected traffic pattern, traffic data is captured and analyzed during a calibration phase (the length of which is specifiable). The boundary controller then starts running statistical hypothesis tests between expected and observed distributions of multiple key metrics to determine whether the amount of outgoing traffic is significantly higher than expected. The distributions' window sizes can be customized to a specific application's traffic patterns, allowing rapid detection of floods. In addition, the time period for which outgoing traffic is rate-limited and the exact maximum rate can easily be specified in the qosket's property file.

Any containment strategy that relies on network sensors to mark other hosts as bad allows the possibility of the attacker fooling the sensors and causing every host to be declared bad and then shut down. To rule out this possibility, we implemented a limit on the number of hosts that could be quarantined based on network sensors. This limit was a fraction of the total number of hosts, but clearly other policies are also possible.

5 Validation

Validation of APOD technology happened in two phases. First, about halfway through the project, the APOD technology was subjected to attacks by a knowledgeable member of the development team. Second, near the end of the project, the APOD technology was subjected to attacks by an outside Red Team. This section reports the results from both phases.

5.1 *In-House Testing*

5.1.1 Experiment Setup

We have used two sample applications for defense enabling in the APOD project namely, *Warfare* and *ATC*. The Warfare application is a small application consisting of 5 components demonstrating the use of an information server and how it could be attacked in a cyber-war. The ATC application is a relatively larger application of 12 components, where multiple sensors report observed information to a fusion component, and a display component presents the fused data in a GUI. Both applications use the same adaptive middleware base to incorporate defense mechanisms.

These applications were hosted on the *APOD testbed*. This testbed is a closed network consisting of two LANs connected by routers. All the hosts in the testbed were running Linux.

The *attacks* on the APOD applications come from an attacker machine within the testbed network. This machine represents a host already compromised by an attacker. The attacker can use this machine for a variety of network attacks including *eavesdropping*, *spoofing* and *flooding* once he has gained root privileges.

The attacker is further given *attacker privileges* to *kill* processes and *reboot* or *halt* computers to simulate further compromises of the APOD computers. The *goal of the internal testing* is not to learn how to gain root privileges on a computer, but rather to understand the impact once an attacker has gained root privileges. Thus we allow the attacker in these tests to have root privileges, rather than forcing him to gain root privileges via some other mechanism.

5.1.2 Attacks

Killing a process/Halting a machine

The attacker can gain root privilege on an application machine and kill the application process or halt the machine.

Intermittent/Continuous Flooding

The attacker can gain root privilege on an application machine and start sending large amounts of traffic to the servers, thereby effectively flooding the network.

Spoofed termination attack

A spoofed termination attack injects a well-formed packet with the FIN flag set. This flag indicates to the receiver that the sender wants to terminate the TCP session. We developed an eavesdropping attack program that observes a CORBA session between a client and a server and used it to attack the warfare example. This program waits for the client to make a request of the server, in this case the information

server (which in reality is a simple count server). It copies the packet sent from client to server and holds it until the server replies. When the server replies, the eavesdropper updates both the sequence and acknowledgement fields of the original packet with new values based on the reply, sets the FIN bit, and sends the packet to the server.

To the server, this packet looks like a real request, with an appropriate sequence and acknowledgement numbers, and with the FIN bit set indicating that the TCP session should be torn down. The server, without the real client's knowledge, then terminates the connection. The next time the client tries to send a packet to the server, it gets an error message back indicating that the server is no longer listening on that port. This forces the client to try to reconnect to the server.

SYN flood DOS attack

The SYN flood denial of service (DOS) attack causes a server to be so overloaded with fake requests for service, that real requests are likely to go unserved. A TCP session has a three-stage start. In the first stage, the initiator sends a TCP packet with the SYN bit set. The receiver of the SYN packet then responds with a packet with both the SYN and ACK bits set. Finally, the initiator responds with a packet that has the ACK bit set, and the connection is established.

When a host receives the first packet, with the SYN bit set, it usually allocates resources for the expected connection, sends the reply, and waits for the third stage packet. But a malicious host can repeatedly send SYN packets, but never send the final ACK packets. This attack causes the target host to have resources allocated against a connection that will never be completed. Because the host is so busy servicing these false SYN packets, it is increasingly likely to miss a real SYN that would establish a proper connection.

The SYN flood DOS attack only serves to disrupt new connections. However, existing connections are unaffected. But when the SYN flood is coupled with the spoofed termination attack, they can be an effective two-pronged attack. The termination attack causes existing connections to be dropped. The SYN flood makes it hard to reconnect. Together, they disrupt both existing and new connections.

Packet replay attack

A packet replay attack injects a well-formed, but misplaced packet into the client/server TCP stream. As in the spoofed termination attack, the eavesdropper captures a client request, waits for the server reply, and then resends the captured packet with updated sequence and acknowledgement numbers. The server will view this packet as valid since it contains a valid CORBA request, so it will respond. But the client will have no knowledge of the request, and thus will not expect the reply with the given sequence and acknowledgement numbers, so it will discard the reply. But when the client does send its next request, the server will have already seen the sequence number indicated, and throw away the packet as having been

received. This effectively destroys the communication between client and server, although it does not shut down the connection outright. From this point on the client and server can not communicate because their TCP session is out of sync.

Man in the middle Attack

The man in the middle attack is a variation on the packet replay attack in which the contents of the original packet are modified slightly. This is an attack against the OO-DTE encryption layer sometimes employed by the Warfare scenario. OO-DTE digitally signs the body of the CORBA message in the TCP packet. When it gets to the other end, the signature is verified. The man in the middle attack modifies the payload of the TCP packet in order to test OO-DTE's ability to detect the modified payload.

Using APOD against itself in the Warfare example

By predicting which port a client will use, and spoofing a flood from the client host and the predicted port, the application can be fooled into blocking the client host and predicted port. The Warfare application uses Snort to detect network floods when it detects that the primary server is no longer available. If Snort detects a flood, ipchains is activated to block the flooder's host and outgoing port so the flood will have no impact on the application. When establishing a client socket, a Unix host uses the next available outgoing port number. We can predict which outgoing port is next by examining the current port that the application is using. Chances are that the next client established will have the port number one greater than the current. The probability is increased if we can cover the next three ports, just in case there are other client applications on this host as well.

5.1.3 Defense Strategies

ATC

The defensive strategy of the application is designed in such a way that repeatedly killing database replicas will eventually push it into a more defensive posture. Once a host has had four failures, the replication management services are exploited to take that host out of the pool of replication hosts. When two hosts have been removed from the pool, and two remain, the ATC example stops launching new replicas and starts to use the Backup Database.

At this point ATC uses Tripwire to periodically monitor the integrity of the file system. If the attacker were to install a Trojan horse program in the ATC directory in the hope of having it run by a privileged user, Tripwire would detect that. If a violation of file system integrity was detected, the defensive strategy of the application is to try to step up the protection of the host that runs the backup database. It achieves this by

using the services of the ipchains Linux firewall mechanism. At this point, the application's security policy is also changed to suspend the administrator's privilege to change the application's tunable parameters.

Warfare

The Warfare example consists of a simple client/server application with client side adaptation. If the client's requests to a server time out, the client will switch over to using a backup server. In addition, it will start listening for abnormal network traffic via snort. Once a network-based attack is picked up, the client will call its local Linux firewall via ipchains to block all traffic to and from the attacker source.

The Warfare defense is limited in terms of defenses and tolerance. Its only defense is the Snort manager and ipchains. Its only tolerance mechanism is a single backup server. Since the backup server is identical to the primary server, any attack that successfully compromises the primary server can be launched against the backup server. There is no fallback once the backup server has been compromised.

5.1.4 Execution & Data Analysis

ATC

Attack	Succeed?	Comments
Killing a process	No	AQuA replication mechanism was able to start a new process within 2 minutes
Halting a machine	No	Ruled out by whiteboard analysis
Intermittent flooding	Yes	Straightforward to eliminate replicated database servers by flooding the network
Continuous flooding	Yes	Straightforward to eliminate replicated database servers by flooding the network

Table 4 - Attacks against ATC

ATC survived 2 out of 4 attacks run against it (see Table 4). Both attacks were able to kill one of the database processes, but the application survived by launching a replica database. Unfortunately, the system seems too vulnerable to network flooding. Ideally, the replicas should be able to continue running while the network is being flooded.

In the case of continuous flooding, the ATC system really needs to change to a different mode. For instance, in the real world, it might have a redundant network available to switch to. A true survivable system would not rely on a single network infrastructure. The APOD test lab does not have the resources to provide multiple networks, so the application's possible defenses are constrained by our test environment.

Warfare

Warfare uses TCP as its underlying communications mechanism. As such, it is open to a variety of attacks that cannot be used against the ATC application. The testing of Warfare is focused primarily on these TCP attacks: All attacks mentioned in the attack chapter were launched against the Warfare application. The attacks were able to interrupt client server communication and have the client fail over to using the backup server. In every case, the application could be blocked by attacking twice, because the backup server is identical to the primary server. For the Man in the Middle Attack, OO-DTE is able to detect the modified packet and reject it, which preserves the integrity of the server. The attack renders the server unusable, however, because the TCP stream has been damaged.

5.1.5 Lessons Learned from APOD In-House Testing

The initial in-house experiments led to the following conclusions about APOD as a defense technology:

- APOD can withstand real attacks. The software testing done beforehand basically simulated certain aspects of attacks, but APOD never had to deal with a real attack. The in-house experiments assured that APOD can be applied to real systems and attacks.
- Replication is useful against host-based attacks, but performs poorly against network-based flooding attacks. The specific replication mechanism used by AQuA relies heavily on UDP multicasts with all or nothing semantics in its higher-level group communication protocols. We observed that flooding, even if confined to a single link in the network, causes major disruption in replication.
- IP spoofing is hard to detect and might lead to self-inflicted denial-of-service. The in-house attackers were able to use APOD against itself by spoofing attack source IP addresses, which led the adaptive firewall reconfiguration to block addresses it is not supposed to block, causing self-inflicted denial of service.

Overall, the results of the in-house experimentation justified further development and validation of APOD; we continued on developing new defense mechanisms and strategies, and finally undertook a formal red team evaluation.

5.2 Red Team Experiments

Two formal red team experiments [50,51] were carried out to evaluate the APOD technology under the umbrella of the FTN/DC continuous experimentation program. The *overall goal* of these experiments was

to investigate and quantify the ability of defense enabling to provide dynamic automated defense of a representative application - an image-serving system - in a realistic, distributed environment. The *application* (Figure 10 depicts a schematic view of the application) used in the experiments is representative of many command and control applications in which data has to be shared between multiple parties in a timely manner. Both clients and servers publish their information needs and information availability to an information broker, which connects clients to servers. Clients then directly interact with the servers to get the desired information. Such interaction patterns are, for example, common in the JBI domain [52]. The first experiment (APOD-1) investigated how defense enabling could help the image serving application survive attacks that damage the broker component. The second experiment (APOD-2) expanded the investigation of APOD-1 to include attacks that attempt to disrupt the image serving application by flooding network links between routers.

5.2.1 Planning

The Red Team validation investigates and quantifies the ability of APOD to provide dynamic automated defense of a representative application – an image-serving system – in a realistic, distributed environment. The validation consists of a set of two experiments: This first experiment, APOD-1, was limited to investigating the use of replication and sensors to detect a subset of damaged components and dynamically replace them with properly functioning ones while under active attack. In the second experiment, APOD-2, defenses against network flooding were incorporated into APOD and attacker constraints were relaxed to test the additional defenses.

5.2.1.1 Staff

Following standard Red Team practices, the experimentation process involved a White Team, a Blue Team, and a Red Team. The APOD project team played the role of the blue team. The FTN/DC continuous experimentation program provided the independent white team and supported the Sandia Red Team for the two APOD red team exercises.

5.2.1.2 Milestones

Table 5 summarizes the major milestones in the red team experimentation process:

Milestone	Completed
Start of experimentation process	Mid Oct 01
APOD-1 whiteboard meeting	Mid Dec 01
APOD-1 lab network established and APOD successfully transitioned	Early Jan 02
APOD-1 experiment plan	Jan 02
APOD-1 execution	Early Feb 02 -Late March 02
APOD-1 hot wash	Late March 02
APOD-2 whiteboard meeting	Mid April 02
APOD-1 Final Report	Mid May 02
APOD-2 experiment plan	Mid June 02
APOD-2 execution	June 02
APOD-2 hot wash	Early July 02
APOD-2 final report	September 02
End of experimentation process	September 02

Table 5 - Red Team Experiment Milestones

5.2.1.3 Whiteboard Results

The white board meeting for APOD-1 was the first meeting where all the groups met in this experimentation process. During the discussions, the various definitions of experiment aspects got significantly refined, including experiment hypothesis, flags, rules of engagement, Red Team logging, and data analysis.

Interactions between Blue Team and Red Team about what the software does and how it could be attacked was very helpful in both understanding attacks and coming up with defenses against them.

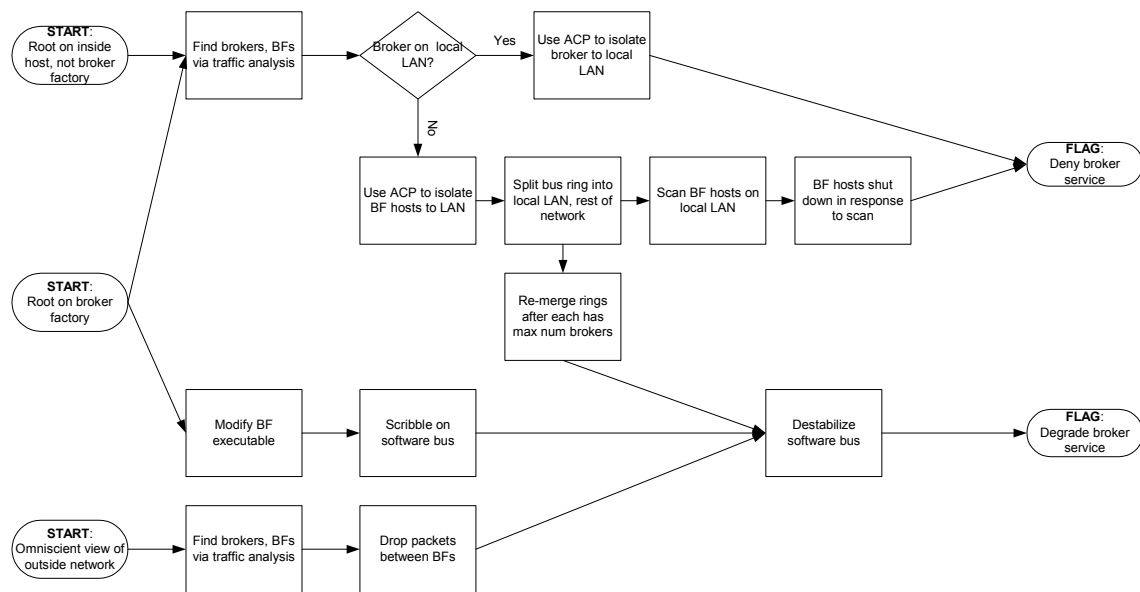


Figure 9 - APOD-1 Attack Plan

As shown in Figure 9, the high level attacks that the Red Team came up with in the whiteboard meeting included:

Deny Service

- Kill brokers
- Prevent packets to or from broker from reaching destination
- Modify broker executable to refuse service
- Break software bus into smaller rings
- Confuse software bus
- Use APOD against itself

Degrade Service

- Cause software bus traffic to clog network
- Prevent software bus from stabilizing (malicious broker factory)
- Confuse software bus

The strategy “Confuse software bus” may cause a denial or degradation of service, or both, or neither. This is why it is listed for both objectives. The experiment was to reveal which objective is achieved by the confusion.

The APOD-2 whiteboard meeting was mainly focused on how the existing definitions and attacks can be changed to test the new defenses against network flooding. The rules of engagement were relaxed and the network topology refined. Compared to the previous whiteboard meeting for APOD-1, the discussion about potential attacks did not seem as fruitful as expected. The Red Team did not change its high-level attack plan for APOD, and only added a plan for two direct attacks on RSVP to a) crash RSVP daemons and b) destabilize through RSVP message spoofing.

5.2.1.4 Network Topology

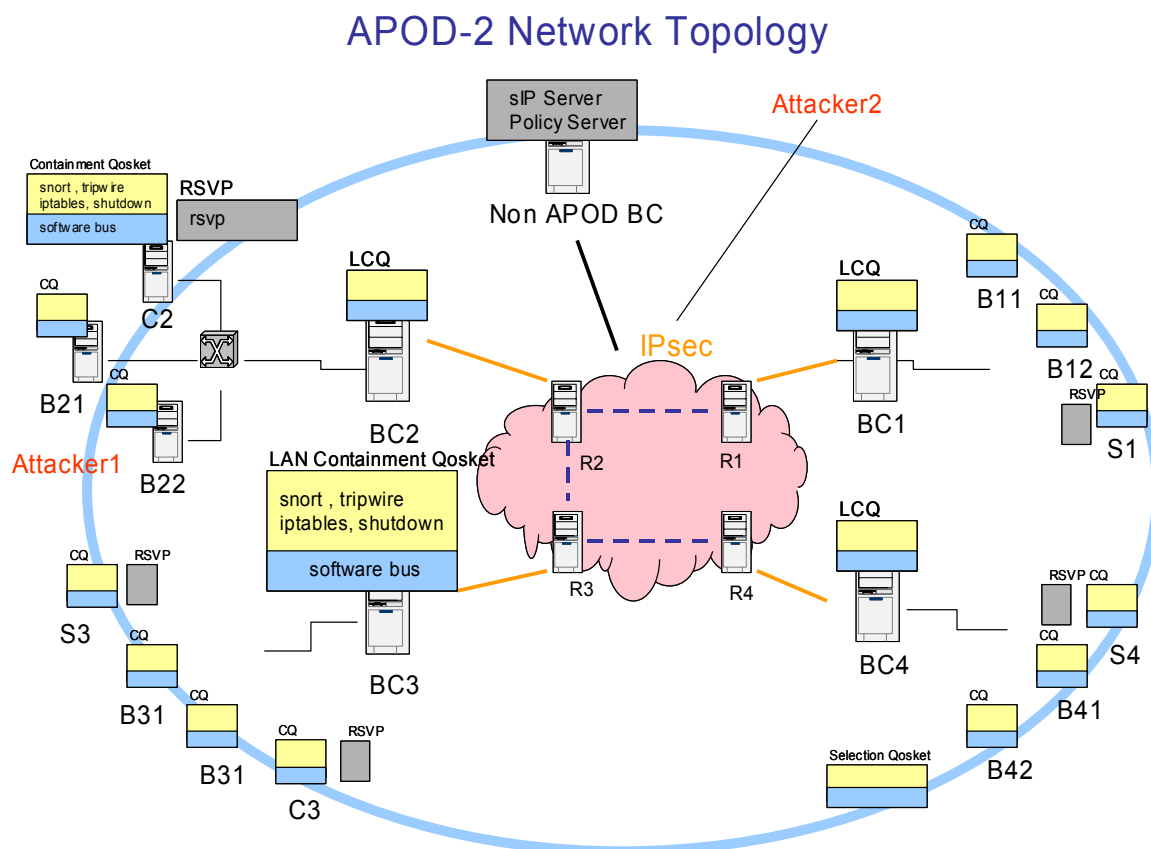


Figure 10 - APOD-2 Network Topology

Figure 10 displays the network topology used in APOD-2.

5.2.1.5 Experiment Hypotheses

The principal, or top-level, hypothesis of the Red Team experiment was:

- APOD improves immunity to cyber attacks relative to systems that do not use APOD.

In addition, the following two specific sub-hypotheses were defined:

- APOD improves immunity to availability attacks relative to systems that do not use APOD.
- APOD improves immunity to availability attacks at an acceptable cost to the defender.

A Red Team experiment can only support or refute the above hypotheses. Because of the scope of the problem space, it is not possible to prove the hypotheses in a definitive manner.

5.2.1.6 Rules of Engagement

The Red Team was given access to all the details of the APOD mechanism, but not necessarily the specific means in which it was used, e.g., the parameters of the defense strategy. In addition, the Red Team was free to use any tools of their choice for implementing the attack. Root access to one of the APOD machines was granted to the Red Team for simulating an insider attack. In addition, access to the central network was allowed for data analysis purposes.

For APOD-1, any attacks pertaining to broker processes were allowed. However, network flooding attacks were excluded, as were attacks prevented by IPsec and OO-DTE (which were assumed but not deployed).

For APOD-2, the Red Team was not allowed to:

- Kill APOD application processes by any means
- Start corrupt APOD application processes
- Damage application components simulating external hardware
- Use the privileged experimental infrastructure (experimentation control and data collection) to implement an attack.

Again, OO-DTE and IPsec were assumed to be in place and working effectively.

5.2.1.7 Metrics

These experiments defined the following principal metrics:

- Image-serving latency, measured across a suite of image clients. The clients will automatically request randomly selected images and capture and log the time between request and completion of rendering. This metric characterizes the degrade flag.
- Viability of the server system - whether or not the attacker has captured the flag (e.g. the fraction of image requests that do not succeed). If the attacker is thwarted due to APOD, then APOD is

considered to have succeeded and to be effective. If the attacker is successful, then APOD is considered to have failed and APOD is considered to be ineffective. This is a binary metric, i.e., there is no intermediate degree of response success or failure.

- Time to denial – how long it takes the attacker to capture the deny flag (e.g. wall clock time). This metric is only meaningful when the deny flag is achieved.

In addition, the principal metrics were augmented by three additional metrics that capture the “work factor” or costs of APOD and the Red and Blue Teams.

- Number of attacker (Red Team) actions. This metric gives an indication of the amount of work required to carry out an attack as well as the amount of risk incurred by the attacker. APOD is considered to be more effective if it causes the attacker to take more actions.
- Number of defensive (Blue Team) actions. This metric gives an indication of the amount of work required to defend against an attack as well as the cost incurred by the defender. APOD is considered to be more costly if it requires a great deal of CPU cycles, bandwidth, extra boxes, testing/data collection, etc. to achieve statistically significant results.
- The average serving latency for the system absent APOD mechanisms versus the average serving latency for the system with APOD mechanisms, but in the absence of attacks. This metric characterizes the direct operational cost of employing this mechanism.

5.2.1.8 Flags

The Red Team could attempt to *degrade* or *deny* availability of requested images to users, using different types of manual and scripted attacks. These two actions were the flags for this experiment:

- In this context, *degrade* implies increasing the latency from the time at which an image is requested until it is rendered, relative to the latency observed in the absence of an attack.
- *Denial* implies that images can no longer be served despite the actions of APOD (e.g., manual intervention is required to restore function, etc.).

5.2.2 Execution

5.2.2.1 Attacks in APOD-1

Name	Attack Strategy Goal	How it works (Method)	Flag Captured?
ScanAll	Shutdown all brokers in one round	NMAP scans using Xmas tree packets and null packets trigger the	No

		IDS	
Scan_Isolate_Flood	<ol style="list-style-type: none"> 1. Shutdown brokers 2. Isolate client and brokers on ipnet 2 3. Add additional obstacle to remaining brokers 	Multi-stage, sequential attack	Yes (Deny)* * 1 time in 3 attempts
Scan_Flood_Isolate	<ol style="list-style-type: none"> 1. Shutdown brokers 2. Interrupt communication between APOD hosts 3. Isolate remaining brokers 	Multi-stage, sequential attack	Yes (Deny)* * 11 times in 18 attempts

Table 6 - Attack Results of APOD-1

The Red Team employed three attacks, “Scan”, “Isolate”, and “Flood” in this experiment, which were used both singularly and in combination. Table 6 displays an overview of the attack runs, while the strategies that lead to the selection of these attacks, and the attack tools used to implement them, are described below.

- Scan – The goal of the scan strategy is to fool brokers and broker hosts into believing that they are being scanned by a compromised broker host. This will trigger the APOD containment defense strategy, because the scanning host is presumed compromised. Hence, this attack turns APOD on itself. The actual scan is launched from one of the attacker’s machines, spoofing (e.g. changing) the source IP addresses of the scan packets to those of a broker host.
- Isolate – This strategy isolates broker hosts on one subnet from all the other subnets. Thus, all brokers and broker factories on the other subnets presume that the brokers and broker hosts on that one subnet have died. All hosts on the same subnet are still allowed to communicate with each other.
- Flood – The goal of this strategy is to prevent the establishment of TCP connections to a specific host/port pair, which interrupts communication between broker factories on the APOD bus, denying broker factory hosts their APOD bus instructions to create new brokers. This strategy requires that the attacker know the ports used by the broker factories for communication over APOD’s self-stabilizing bus. This information is determined either by traffic analysis, or by decoding the IORs (CORBA Interoperable Object References) on the broker factory hosts (which requires access to those hosts). This strategy is not precluded by the use of OO-DTE, because a TCP connection must be established before it is authenticated.

These strategies were used singularly and in combination as specific attacks, as named below, respectively. The goal of, and logic behind, the construction of each attack is pointed out.

- Scan-All attack – This attack makes APOD believe that all broker factory hosts are scanning. The goal of the attack is to make all APOD hosts shut down in a single round of APOD bus communication.
- Isolate and Scan attack – This attack occurs in two consecutive stages; first isolating the broker factories on IPNET2; then spoofing APOD into believing that the remaining broker factory hosts are

scanning. The goal of the attack is to first shut down the IPNET2 hosts by isolating them, then to shut down as many of the remaining hosts as possible by spoofing that they are scanning.

- Scan, Isolate, Flood attack – This attack occurs in three consecutive stages, where each stage prevents some of the broker factory hosts from fulfilling their roles as a broker. First, the Scan attack is directed at four broker factories, which include the two active brokers (determined by traffic analysis). This attack causes APOD to kill all four broker factories. Second, the broker factories on IPNET2 are isolated from the APOD bus (in all the test runs, the initial brokers were not on IPNET2), which leaves two broker factories available to be commissioned as brokers. Third, to prevent that from happening, the Flood attack is applied to those two broker factories, so that the APOD bus believes all other broker factories are dead. Hence no new brokers are created, and image service is denied.
- Scan, Flood, Isolate attack – This attack reverses the order of the last two stages of the previous attack. As the previous attack did not succeed in all the experimental runs, it was speculated that APOD's response was sufficiently fast enough, so that sometimes the last two broker factories were commissioned as brokers before the Flood attack took effect. Therefore, this switch was made, so that the Flood attack was begun just before the Isolate attack.
- Scan, TCPKill, Isolate attack – This attack is a variant of the Scan, Flood, Isolate attack, where the Flood attack is replaced by the TCPKill attack.

5.2.2.2 Attacks in APOD-2

Name	Attack Strategy Goal	How it works (Method)	Flag Captured?
localhost_single_packet	Trigger Snort	Sends single, benign packet with a source address of 127.0.0.1 => propagates iptables rule	Yes (Deny)
localhost_scan_bc	Trigger Snort	Spoofs scan from outside interface of boundary controller on the inside interface of the same bc => propagates iptables rule	Yes (Deny)
localhost_scan_any	Trigger Snort	Spoofs scan with source address of 127.0.0.1 => propagates iptables rule even if snort is configured to ignore localhost rule.	Yes (Deny)
spoof_bh_block_bc_spoof_lan	Trigger Snort Sever Bus	multi-stage sequential attack	Yes (Deny)
spoof_bh_block_bc_spoof_lan_fast	Same as previous, but with reduced wait times	multi-stage sequential attack	Yes (Deny)
Tcpjunk	Disrupt/Deny APOD communications	Attempts to insert traffic in the TCP stream in order to force the TCP connection to re-sync or fail.	No
Tcpkill	Disrupt/Deny APOD communications	Sends TCP packets with reset flag set to terminate connections	1/2 Deny (when client was on)

			attacker's subnet)
Tcpreplay	Interfere with legitimate RSVP traffic Confuse RSVP nodes	Using captured RSVP traffic, traffic is replayed on the network from the attacker's host	Yes (degrade)
rsvp_fuzzer	Crash RSVP daemons Interfere with legitimate RSVP traffic	Inject crafted bogus RSVP packets, including teardown packets.	No

Table 7 - Attack Results of APOD-2

Two principal attack strategies were pursued in the second APOD experiment:

- Using APOD's own defensive mechanisms against itself
- Attacking SE-RSVP, an underlying service APOD uses to guarantee bandwidth

Attacks were employed both singly and in combinations where it was deemed logical and necessary for the Red Team to accomplish its objectives. Table 7 gives an overview of the results.

Using APOD against itself:

- Trigger Snort Violations - spoofed port scans using Xmas tree and null packets were effective at triggering the IDS and propagating *drop traffic* rules among the APOD hosts. Using such scans generally resulted in multiple APOD hosts shutting down. Multiple attacks were developed that exploited *local host* traffic, resulting in a nearly instantaneous shutdown of all APOD broker hosts.
- Sever the APOD Bus - isolating the broker hosts on one subnet using ARP Cache Poisoning resulted in additional broker host deaths.
- Disrupting APOD Communications - TCP connection floods to interfere with legitimate traffic (used successfully in APOD-1 against the broker hosts) were ineffective in APOD-2. Traffic additions did get iptables rules propagated but, without additional methods, failed to capture a flag. TCP connection resets were effective where an APOD client was on the attack machine's subnet and traffic encryption was not present.

Attacking SE-RSVP:

Crash RSVP Daemons - injecting bogus RSVP traffic into the stream had intermittent results. Even when RSVP daemons did crash, the reservations were unaffected as they are static and assigned upon system initialization.

Interfere With Legitimate RSVP Traffic - use of injected, crafted RSVP traffic, and authentic RSVP traffic replay was ineffective except when the introduced traffic was flooded.

Confuse RSVP Nodes - faking RSVP teardowns to terminate bandwidth guarantees were not achieved, due to the cryptographic authentication utilized in SE-RSVP.

5.2.3 Data Analysis of Primary Metrics

The attacks performed in APOD-1 were mostly aimed at the defense strategies: the Red Team attempted to use APOD against itself to capture the flags. Table 7 describes the attacks and their outcomes. To study the performance of the defense-enabled application, the White Team performed elaborate baseline and engineering tests. In addition, completely scripted attack runs were executed to gather statistical data. Although the Red Team was able to capture the denial flag most of the time, the experiment showed that APOD improved immunity at an acceptable cost. APOD's mechanisms had little operational overhead in APOD-1 (5% additional latency). In order to capture the flags, the attack strategies required multiple phased attacks that increased the Red Team's preparation time and efforts. The Red Team had to do more system research and spent more time developing exploits. The reconnaissance required to study the defense-enabled system was time consuming. APOD's replication management capability (coupled with a dynamic firewall) proved to be an effective defense against several well-known attacks, both individually and in combination, on image brokers. More complex or persistent attacks targeted at APOD were successful.

The focus of APOD-2 was to evaluate the APOD defense against network-based flooding attacks and the impact on adding this new survivability requirement on to those of APOD-1. In other words, the Red Team rules of engagement for APOD-1 were further relaxed to allow flooding of network links between routers. Therefore, a new defense strategy called *Flood prevention and trace back* was developed and deployed. This strategy statically established a set of RSVP reservations at startup, using SE-RSVP. The purpose of the RSVP reservations was simply to prioritize application traffic and traffic from the APOD components, allowing the system to tolerate network floods. In addition, the strategy contained coordinated defense behavior implemented via a boundary controller on each LAN. Upon detecting a significant increase in outgoing traffic, this mechanism would rate-limit outgoing traffic for a certain amount of time. This new LAN containment mechanism was also interfaced with the already existing host quarantine logic of APOD-

1 to implement a trace back capability: Suspicious hosts on APOD-protected LANs were blocked closest to their source. The network topology of APOD-1 was modified to accommodate the router requirements necessary to implement the defenses against network flooding (see Figure 10 for a pictorial representation of the network topology). Also, the Red Team rules of engagement were modified to make sure the Red Team would not exploit vulnerabilities SE-RSVP did not address. While the new defense strategy and mechanisms used for APOD-2 defended against new attacks, this improvement came with a cost: the overhead imposed by APOD-2 increased to roughly 24% (from 5% in APOD-1).

Defense enabling clearly raised the bar for the attacker. Without APOD defense enabling, the image server application would have at least one single-point-of-failure. For example, killing a single application process would lead to capture of the denial flag. An attacker starting with “root” privilege on one host could therefore capture this flag immediately. In contrast, when the Red Team started to attack the defense-enabled application with “root” privilege on a single host, capturing the denial flag took significant amounts of time. Figures 11 and 12 show measurements of this time for various Red Team attacks. In Figure 11, the average time to denial was 44.4 minutes for live attacks and in Figure 12, 18.8 minutes was the average for scripted attacks (vs. essentially zero for the non-defense-enabled application).

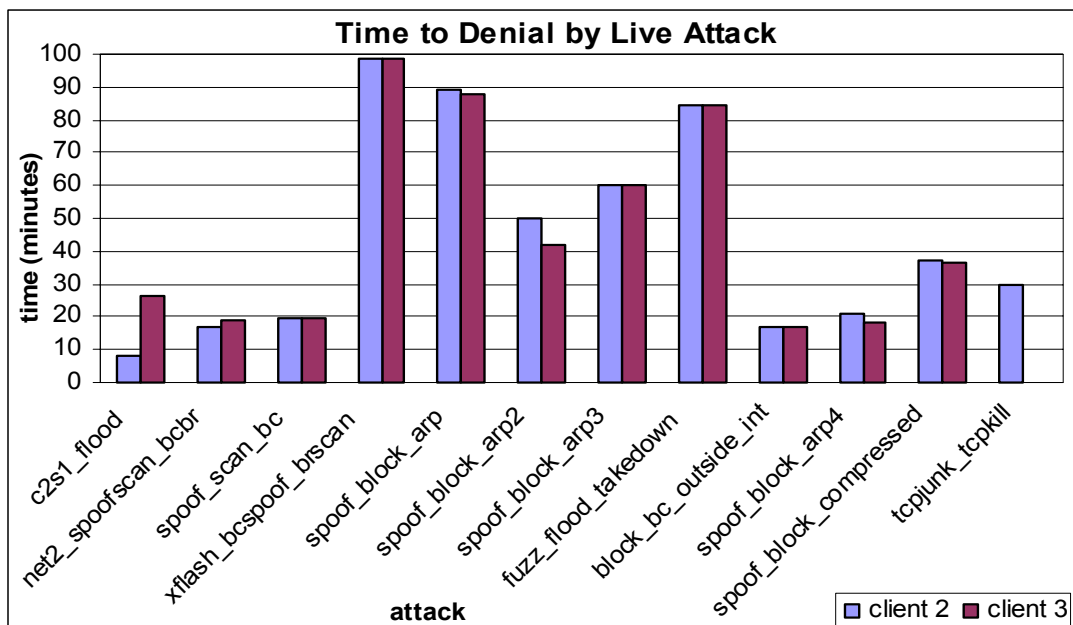


Figure 11 - Time to Denial for Live Attacks

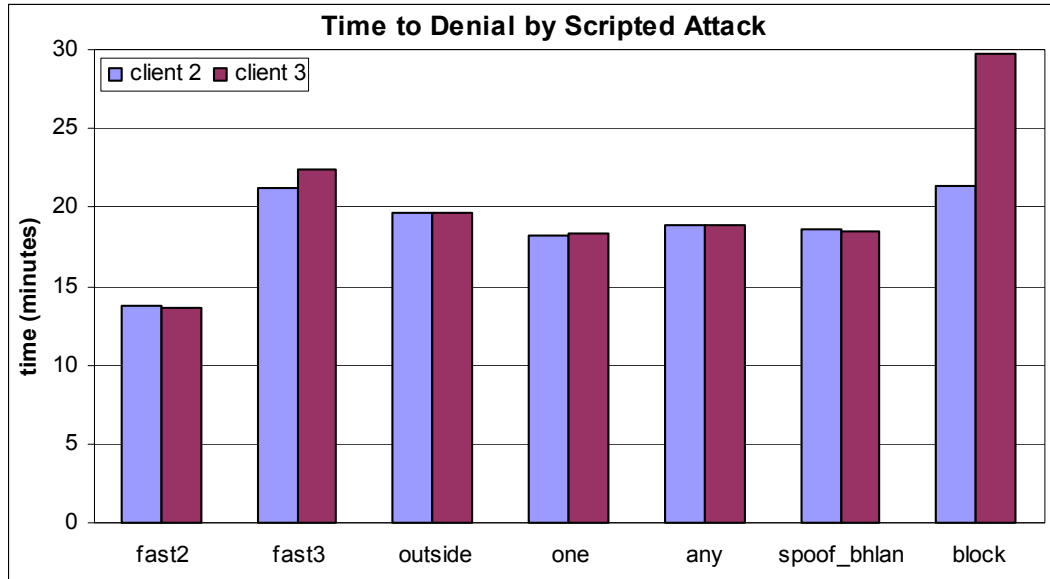


Figure 12 - Time to Denial for Scripted Attacks

Another interesting conclusion was that in most cases APOD-2 stood its ground against both live and scripted attacks until the very last image broker died. As an example, see the plot of image latency measurements displayed in Figure 13 for a particular attack run. Although this plot shows some outliers, i.e., measurements in which latency was significantly more than average, overall this plot shows that image latency does not change significantly over time, despite ongoing attacks by the Red Team, until the application finally dies 3200 seconds into the run. This suggests that the overhead introduced by APOD while defending the application is approximately the same as the runtime overhead in absence of any attacks. This property of APOD can be viewed as desirable: APOD kept the application's performance from degrading as long as possible. On the other hand, this fact opens another question: would a different defense strategy allow greater survivability with graceful degradation?

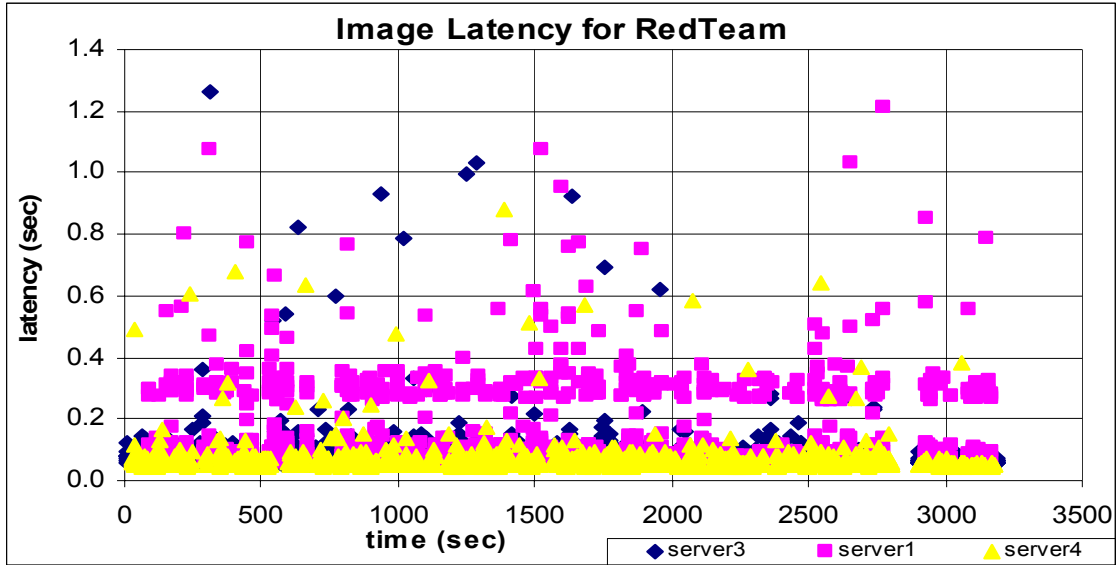


Figure 13 - Attack spoof_block_arp4

5.3 Discussion

This section first contrasts the lightweight in-house Red Team experiment with the formal and extensive Red Team exercises, and then reflects on designing next-generation experiments for evaluating research grade and evolving survivability solutions based on the insights we gained through the initial set of evaluation experiments.

5.3.1 Contrasting In house and Formal Red Team Evaluations

The focus of the *in-house testing* was primarily on gaining some understanding whether or not our rapid prototype software can be used to stay survivable in the presence of real attacks. One person was assigned the role of a mini Red Team, who developed high level attacks (see Section 5.1.2) and executed them in a loosely specific testbed. The testing was executed with the workload of 2 man months. We perceived many useful insights into various attacks gained during this testing. In addition, the execution of some of the attacks pointed out weaknesses in APOD that were addressed in later development cycles. We see in-house testing as a lightweight but effective way to get early feedback about usability of rapid prototyped software.

In contrast, *formal Red Team testing* was performed at the end of the APOD project. The goals were to validate the APOD technology in a more thorough way and transition the APOD software into a form where it could be experimented with by an independent team. APOD was subjected to 2 Red Team experiments, which were organized in a systematic way. Each Red Team experiment required a considerable amount of work due to whiteboard meeting reports, experiment plan, hot wash reports, and final reports for each experiment. Overall, this evaluation demonstrated that the idea of defense enabling

shows promise in enhancing the survivability of critical applications both in terms of making the attackers work harder to capture the flags and in its ability to quickly respond to the flaws uncovered by the Red Team. The latter goes a long way in making the prototype implementation of the underlying technology even more robust for the next round of experimentation

5.3.2 Ideas for Next Generation Evaluation Experiments

The initial set of formal Red Team experiments were very important milestones in the evolution of defense enabling as a concept and the underlying technologies supporting it. However, there are some aspects of defense enabling that were not stressed in the initial two experiments, but are still in need of evaluation from the APOD developers' perspective. To begin the discussion on next generation of evaluation experiments, recall the White Team's conclusion about APOD-1 and APOD-2. In [50, 51, 55] it was concluded that defense enabling raised the bar for the attacker, but did not prevent more complex, intelligently targeted attacks. To put this in context, note that the goal of defense enabling is not necessarily to *prevent* flag capture (although if it does prevent that, so much the better). Defense enabling is a survivability technology whose goal is to prolong the useful life of the application long enough in the face of attacks. From our perspective, it is important to also focus the evaluation on duration of useful operation and its *incremental improvement*. In that sense, our primary goal is to see if a defense-enabled distributed application can withstand and survive red team attacks *better* than the undefended application and to *quantify* the incremental improvement and its associated cost. The design of a next generation of experiments might differ from the recently completed set of experiments in a number of ways.

First, we expect that the strategy and mechanisms to be used in defense enabling and the Red Team flags chosen would depend upon the survivability requirements of the application. In APOD-1 and APOD-2, the defense (strategy and mechanisms) and Red Team flags were not derived this way. Instead of looking at the survivability requirements first, a subset from the set of the most mature and robust defense mechanisms were chosen to experiment with. This excluded the use and deployment of some mechanisms as well, on the grounds that the benefits and use of these are well known and therefore there is no need to spend resources to further experiment with them. Although this was consistent with the experimentation paradigm and was motivated in part to provide the maximum benefit within the resource constraints, it did impose artificial and synthetic defense strategies, required complicated rules of engagement, and negated evaluation of synergies between mechanisms. This led to instances where the rules were ill-specified or unclear, which in turn lead to problems in their enforcement. In one instance, an attack, which would have likely failed due to IPsec, was devised and executed successfully. In another instance, attack scripts were based on TCP port information obtained through the (unprotected) experiment control infrastructure.

Second, we expect that attacks would be launched on both (A) the raw (undefended) and (B) the defense-enabled application, and the data obtained from the various runs of both these cases will be used to construct A/B comparisons. Selected metrics would be used to assess whether the defense-enabled application survived better than the undefended one, and at what cost. One metric that is useful in this regard is the *Total Time from start of attack to flag capture*. To make the A/B comparison with respect to this metric more meaningful, the Red Team may try to attempt to launch the “best” possible (under the rules of engagement) attack they can on both the (A) and (B) cases, where the definition of “best” may mean, for example, the *fastest* or *stealthiest* way to capture the flag. Techniques based on pairwise comparisons of means [54] might be used to assess the performance increases with respect to prolonging the useful lifetime of an application under attack. In addition metrics like Red Team work-factor can also be used in the A/B comparison. Red Team work-factor has been used in other experimentation programs [58]. Although often criticized as a measure that depends on the specific Red Team's skills, it may still provide valuable insight into how well the defense held up against the attacker, if it is used judiciously in conjunction with a well-defined attacker model.

It should be noted that the usefulness of A/B comparison of Red Team work-factor was discussed during the planning phase of the APOD experiments. It was concluded that the attacks are likely to be different for the raw and the defense-enabled applications as most of the attacks on the defense-enabled applications tried to use the defense against itself. Therefore, such A/B comparisons were not straightforward and not performed. However, as shown in similar contexts in the fields of psychology and artificial intelligence [54], there are statistical methods that may be applied even if the attacks are different. Next generation evaluation experiments might compare the defense-enabled application with the undefended application in terms of a number of quantities like Red Team actions, time to denial, and time spent revising and refining⁵ the attack.

In APOD-1 and APOD-2 the effectiveness of APOD's defenses were summarized in terms of percentage of successful attacks [50, 51, 55]. This metric can be somewhat misleading, since it does not take the nature and coverage of the attacks into account. For instance, [51] reports that the Red Team consistently achieved the deny flag for a large fraction (approximately 3/4th) of the 26 attacks runs in APOD-2. Studying the results of the attacks in detail after the experiments, we observed that the successful attacks fall into one of the following two categories:

- Attacks based on spoofing localhost / boundary controller addresses resulting in self-inflicted denial of service
- TCP connection floods against application processes

⁵ This refers to the time the Red Team prepares to overcome the unexpected defensive responses during attack, as opposed to the planning and initial development time.

This insight suggests that the percentage measure may be obscuring only a single (or a few) issues which are just variations of a theme. Finer grained metrics may be in order.

Third, additional metrics need to be considered for future experiments. APOD-1 and APOD-2 focused almost exclusively on call latency and runtime overhead with respect to call latency. (Time to denial was added in APOD-2, and was later retrofitted on APOD-1 data.) Latency is useful in understanding the operational aspects of the application. However, other measures are also necessary to understand the effectiveness of adaptive defense against the attacks. Examples of other metrics that will provide insight in this regard include: Number of attacker (Red Team) actions, Number of defensive (Blue Team) actions, number of Red Team decision/plan changes, and noise level (indicating how stealthy the attack is) of attacks. The defense could be considered to be more effective if it forces the attacker to take more actions. The number of defensive actions gives an indication of the amount of work required to defend against an attack as well as the cost incurred by the defender. Decisions made by the Red Team during an attack are often significant and worthy of being reported, even if they result in some actions not being taken. Abandoned attack paths can be used to get an estimate of the coverage of the defense. Corresponding graphs would also make the attack model explicit by shedding light on the Red Team's reasoning during execution. The attack stealth is also worth reporting: less-noisy attacks will likely be more effective against automated defenses that have no human defender in the loop.

Fourth, it is becoming common to stipulate survivability requirements in terms of preventing a certain percentage of attacks from achieving attacker objectives for a certain time period [59]. Another way to formulate the next-generation experiments is to define which attacks will be launched upfront, and setting up time bounds for survival and metrics which indicate how closely these goals are achieved.

Finally, we would like the next-generation Red Team experiment processes for evaluating survivability to be cognizant of the fact that technologies like defense enabling are in reality research prototypes and not finished products. The two APOD Red Team experiments were quite effective in transitioning our rapidly prototyped software from a capability under development to one suitable for laboratory experimentation. However, a methodology which provided for fixing simple implementation deficiencies of rapidly evolving technology during experimentation would permit more insight about the underlying concepts. We therefore advocate smaller, more regular lightweight Red Team experiments to continuously evaluate research software throughout its project life cycle. We also believe that doing an internal Red Team test before beginning more formal Red Team experimentation is a useful practice. For APOD, prior to APOD-1 and APOD2, lightweight internal testing was done six months into the APOD project [19]. The internal evaluation provided insights as to whether APOD could be used successfully to defend against real Red Team attacks as well as providing useful feedback for improving the underlying technology.

6 Concluding Remarks

The APOD project has defined and developed a method, called *defense enabling*, for making distributed systems resilient to attack. Defense enabling is representative of a relatively recent trend in computer security, often called *survivability* or 3rd generation security. The problem addressed by this trend is not new, as can be seen in the following synopsis of the 3 generations:

- 1st generation security aimed to protect systems with software and hardware mechanisms that could not be circumvented. This approach has proved to be quite costly and inflexible.
- 2nd generation security acknowledged that implementations of 1st generation protection are imperfect, allowing attackers to circumvent security mechanisms. The 2nd generation aimed to detect attacks, allowing system operators to respond.
- 3rd generation security uses both 1st and 2nd generation techniques but adds automated defenses that respond to attacks. These defenses can respond to the *effect* of attacks even when the attacks themselves are not detected with 2nd generation techniques.

The 3rd generation has proved to be necessary in modern computer systems, which typically rely on COTS components whose security characteristics are not known and that interconnect and interoperate with many other systems. The use of insecure COTS means that the 1st generation approach is impossible. The openness of these systems results in the constant discovery and use of new attacks. When 2nd generation techniques fail to detect these new attacks, 3rd generation techniques are necessary.

Several factors distinguish our approach to survivability from others. First, dynamic adaptation is one key theme of our approach. Intrusions cause changes in the system, and a survivable system must cope with these changes. As a consequence, defense-enabled applications must be very agile and will make use of the flexibility possible in today's dynamic, networked environments. Second, a defense-enabled application has a defense strategy that is typically application- and mission-specific. Such strategies complement and go beyond traditional approaches to security in which protection mechanisms are typically not aware of the applications they aim to protect. Third, defense enabling builds the defense in middleware, intermediate between the application and the networks and operating systems on which the application runs. Recent developments in middleware allow us to develop adaptive defenses quickly. Advanced middleware allows us to base these defenses on information from layers both above and below, i.e., from the application and from the infrastructure, as well as allowing us to control and dynamically adjust aspects of each layer. Defense strategies implemented in middleware can be reused relatively easily in the context of other applications because they are only loosely coupled to the application.

Each of the factors distinguishing defense enabling from other approaches has posed ongoing research challenges. For example, the space of defense strategies has only begun to be explored. As another example, strategies should be structured for quick reuse in many applications. This structuring may lead to

extension of the mechanisms underlying the defense strategy, possibly pushing some defense capabilities from the middleware layer into lower-layer mechanisms.

The APOD project gave promising results but these results are neither conclusive nor complete. As described in Section 2.4, defense enabling has been shown to force attackers, even highly skilled ones, to work harder and longer to damage a defense-enabled application. However, we believe that these results are still a factor of 3 or 4 short of providing practical survivability. We also believe that our results are the state of the art: we are unaware of other research that has demonstrated greater survivability. Therefore, our immediate goal is now to improve the defense enabling approach to gain the missing factor of 3 or 4, thus establishing that it is possible to build survivability into an application.

We expect that we can show that survivability is practical in the near future in ongoing projects at BBN. In the ITUA project we are developing new technology for defense enabling that will eventually include all the APOD defense mechanisms and others. The ITUA project also concentrates on defense strategies that offer unpredictable variation, making successful attacks harder. In the DPASA project we are exploring architectural solutions to improve the survivability of mission critical systems. The DPASA technology, and possibly the ITUA technology also, will be subject to Red Team experiments, as APOD was, and in those experiments we will learn whether our newer defenses will lead to increased survivability.

Once survivability has been conclusively shown to be practical, many other research directions will need to be explored. The defense strategies we have implemented so far are suitable for the simplest kind of survivability requirement, in which an application must survive attacks for some period of time. The requirements for fielded systems are likely to be more complex, including the need to trade QoS of one part of the system for QoS in another part, or to trade one kind of QoS for another. With such requirements, the defense strategy will become more involved and new capabilities must be built to support it. Also, a more complex defense strategy is likely to be built from several sub-strategies. Further research will be needed to learn how best to specify individual strategies, their assumptions and dependencies, how to compute the properties of the complex strategy that results from the combination of individual strategies, and how to identify and resolve contradictions between the different strategies being combined.

7 References:

- [1] Xtradyne's Homepage: <http://www.xtradyne.de/>
- [2] Tom Markham, Charlie Payne, "Security at the Network Edge: A Distributed Firewall Architecture", DISCEX-II 2002
- [3] D. Kewley, R. Fink, J. Lowry, M. Dean, "Dynamic Approaches to Thwart Adversary Intelligence Gathering", DISCEX-II 2002
- [4] Netfilter Homepage <http://www.netfilter.org>
- [5] Linux Advanced Routing & Traffic Control HOWTO <http://www.tldp.org/HOWTO/Adv-Routing-HOWTO/>
- [6] Iproute2 Documentation <http://defiant.coinet.com/iproute2/>
- [7] Cisco Intrusion Detection Planning Guide <http://www.cisco.com/univercd/cc/td/doc/product/iaabu/idpg/>
- [8] Cisco IOS Firewall http://www.cisco.com/warp/public/cc/pd/rt/2600/prodlit/flrrr_ov.htm
- [9] Zebedee Secure IP Tunnel <http://www.winton.org.uk/zebedee/>
- [10] FreeS/WAN project homepage <http://www.freeswan.org>
- [11] IETF RFC 2401 Security Architecture for the Internet Protocol
- [12] Detailed IPsec information
 - ietf RFC 2085 HMAC-MD5 IP Authentication with Replay Prevention
 - ietf RFC 2104 HMAC: Keyed-Hashing for Message Authentication
 - ietf RFC 2202 Test Cases for HMAC-MD5 and HMAC-SHA-1
 - ietf RFC 2207 RSVP Extensions for IPSEC Data Flows
 - ietf RFC 2403 The Use of HMAC-MD5-96 within ESP and AH
 - ietf RFC 2404 The Use of HMAC-SHA-1-96 within ESP and AH
 - ietf RFC 2405 The ESP DES-CBC Cipher Algorithm With Explicit IV
 - ietf RFC 2410 The NULL Encryption Algorithm and Its Use With IPsec
 - ietf RFC 2451 The ESP CBC-Mode Cipher Algorithms
 - ietf RFC 2521 ICMP Security Failures Messages
- [13] Openssh <http://www.openssh.org>
- [14] Cisco IPSEC http://www.cisco.com/cgi-bin/Support/PSp/psp_view.pl?p=Internetworking:IPSec
- [15] ATC VPNshield <http://www.atcorp.com/products/vpnshield/vpnshield.htm>
- [16] Janet Lepanto, William Weinstein, Draper Laboratory, CONTRA - Camouflage of Network Traffic to Resist Attack DARPA OASIS PI Meeting, Hilton Head, March 12-15, 2002
- [17] CERT Advisory CA-1996-21: TCP SYN Flooding and IP Spoofing Attacks
<http://www.cert.org/advisories/CA-1996-21.html>
- [18] D.J. Bernstein, "SYN Cookies", <http://cr.yp.to/syncookies.html>
- [19] Extreme Programming Rules <http://www.extremeprogramming.org/rules.html>
- [20] Diffserv Website <http://www.ietf.org/html.charters/diffserv-charter.html>
- [21] RSVP Website <http://www.isi.edu/div7/rsvp/>
- [22] Darmstadt RSVP implementation <http://www.kom.e-technik.tu-darmstadt.de/rsvp/>
- [23] North Carolina State University Secure-RSVP implementation <http://arqos.csc.ncsu.edu/>
- [24] DIRM website <http://www.dist-systems.bbn.com/projects/DIRM/>
- [25] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz. "AQuA: An Adaptive Architecture That Provides Dependable Distributed Objects". Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98), West Lafayette, Indiana, USA, October 20-23, 1998, pp. 245-253.
- [26] M. K. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in Rampart". In Proceedings of the 2nd ACM Conference on Computer and Communication Security, pages 68-80, November 1994.
- [27] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith, "The SecureRing Protocols for Securing Group Communication", Proceedings of the IEEE 31st Hawaii International Conference on System Sciences, Kona, Hawaii (January 1998), vol. 3, pp. 317-326.
- [28] Eternal Project Web Page: <http://beta.ece.ucsb.edu/eternal/Eternal.html>
- [29] APOD Toolkit Documentation, <http://apod.bbn.com/release/latest/ApodToolkit.htm>

- [30] D. F. Sterne, G.W. Tally, C. D. McDonell, D. L. Sherman, D. L. Sames, P. X. Pasturel, and E. J. Sebes, "Scalable access control for distributed object systems", Proceedings of the 8th Usenix Security Symposium, August 1999.
- [31] IETF RFC 2205 Resource ReSerVation Protocol (RSVP)
- [32] S. Staniford-Chen, B. Tung and D. Schnackenberg, "The Common Intrusion Detection Framework", Proceedings of the Information Survivability Workshop, October, 1998.
- [33] P.G. Neumann and P.A. Porras. "Experience with EMERALD to date", Proceedings of the 1st Usenix Workshop on Intrusion Detection and Network Monitoring, April 1999.
- [34] S. Staniford-Chen, S.Cheung, R.Crawford, M.Dilger, J.Frank, J.Hoagland, K.Levitt, C.We, R.Yip, and D.Zerkle. "GrIDS" - A graph based intrusion detection system for large networks. Proceedings of the 19th National Information Systems Security Conference, September 1996.
- [35] S. Stolfo, A. Prodromidis, S.Tselepis, W. Lee, D. Fan, and P.Chan. "JAM: Java agents for meta learning over distributed databases". Proceedings of KDD-97 and AAI 97 Workshop on AI Methods in Fraud and Risk Management, 1997.
- [36] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks", Proceedings of the 13th the USENIX System Administrators Conference (LISA '99), November, 1999.
- [37] G. Kim and E. Spafford. "The design and implementation of Tripwire: A filesystem integrity checker". Proceedings of the 2nd ACM Conference on Computer and Communications Security, 1994.
- [38] J.A. Zinky, D.E. Bakken, and R.E. Schantz. "Architectural support for Quality of Service for CORBA objects". Theory and Practice of Object Systems, 1(3):55--73, April 1997
- [39] R. Vanegas, J.A. Zinky, J.P. Loyall, D.Karr, R.E. Schantz, and D.E. Bakken. "QuO's runtime support for quality of service in distributed objects". Proceedings of Middleware 98, the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing, September 1998.
- [40] P. Pal, F. Webber, R. E. Schantz, M. Atighetchi and J. P. Loyall. "Defense-Enabling Using Advanced Middleware- An Example". Proceedings of MILCOM 2001, Tysons Corner, VA, October, 2001.
- [41] F. Webber, P. Pal, R. E. Schantz and J. P. Loyall. "Defense-Enabled Applications". Proceedings of DISCEX II, Anaheim, CA, May 2001.
- [42] US Department of Defense, "Trusted Computer Evaluation Criteria (Orange Book)", December 1985, DoD 5200.28-STD.
- [43] B. Schneier, "Applied Cryptography", John Wiley & Sons, 1996.
- [44] S. Kent, "On the Trail of Intrusions into Information Systems", IEEE Spectrum, December 2000.
- [45] D. Bakken, "Middleware", <http://www.eecs.wsu.edu/~bakken/middleware-article-bakken.pdf>.
- [46] M. Barborak, M. Malek, and A. Dahbura, "The Consensus Problem in Fault-Tolerant Computing", ACM Comp. Surv., 25(2), 1993.
- [47] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", ACM Comp. Surv., 22(4), Dec 1990.
- [48] The APOD Toolkit Open-Source Release, <http://apod.bbn.com/release/latest>
- [49] Intrusion Tolerance by Unpredictable Adaptation, <http://itua.bbn.com>
- [50] Nelson B., Farrell W., Atighetchi M., Kaufman S., Sudin L., Shepard M., Theriault K., "APOD Experiment 1 - Final Report", BBN Technologies Technical Memorandum 1311, May 2002
- [51] Nelson B., Farrell W., Atighetchi M., Clem J., Sudin L., Shepard M., Theriault K., "APOD Experiment 2 - Final Report", BBN Technologies Technical Memorandum 1326, September 2002
- [52] United States Air Force Scientific Advisory Board, *Report on Building the Joint Battlespace Infosphere, Volume 1: Summary*. SAB-TR-99-02. 1999
- [53] Cukier M., Courtney T., Lyons J., Ramasamy H.V., Sanders W.H., Seri M., Atighetchi M., Rubel P., Jones C., Webber F., Pal P., Watro R., Gossett J., "Providing Intrusion Tolerance with ITUA", Supplement of the 2002 International Conference on Dependable Systems and Networks, June 23-26, 2002
- [54] Cohen P.R., *Empirical Methods for Artificial Intelligence*. MIT Press, 1995
- [55] Webber F., Reeves D., Nelson B., Clem J., "APOD Experimentation Results", presented at the Fault Tolerant Networks PI Meeting, Newport RI, July 2002
- [56] Gerald W. Gordon, GSEC Practical Assignment version 1.3, "SYN Cookies: An Exploration", http://www.giac.org/practical/Gerald_Gordon_GSEC.doc
- [57] R. Schantz, J. Loyall, M. Atighetchi, P. Pal, "Packaging Quality of Service Control Behaviors for Reuse", IEEE Int'l Symp. Object-Oriented Real-Time Distributed Comp., April 2002

- [58] Kewley D., Fink R., Lowry J., Dean M., "*Dynamic Approaches to Thwart Adversary Intelligence Gathering*", Proceedings of the second DARPA Information Survivability Conference and Exposition (DISCEX II), June 12-14, 2001, Anaheim, California.
- [59] The OASIS Dem/Val RFP, http://www.darpa.mil/ipto/Solicitations/PIP_02-16.html